Virtualization Extensions Architecture Specification

ARM Architecture Group

Document number:	Derived from: **PRD03-GENC-008353 14.0**
Date of Issue:	22 October 2010
Author:	ARM Limited
Authorised by:	

In this document, where the term ARM is used to refer to the company it means “ARM or any of its subsidiaries as appropriate”.

Note

The term ARM is also used to refer to versions of the ARM architecture, for example ARMv6 refers to version 6 of the ARM architecture. The context makes it clear when the term is used in this way.

Contents

1 ABOUT THIS DOCUMENT

The material in this document will be incorporated in the next release of the ARMv7-AR Architecture Reference manual (Revision C).

It is at a Beta specification state. Please communicate any errata or issues to errata@arm.com.

1.1 Change history

This document is derived from an ARM internal document, PRD03-GENC-008353 14.0, and is supplied as PRELIMINARY INFORMATION about the ARMv7-A Architecture Virtualization Extension announced in August 2010.

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
1	ARM DDI 0406 B	ARM Limited	ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition
2	PRD03-GENC-008469	ARM Limited	Large Physical Address Extensions Specification
3	ARM IHI 0048	ARM Limited	ARM Generic Interrupt Controller
4	ARM DDI 0457	ARM Limited	ARM Architecture Reference Manual Performance Monitors v2 Supplement
5	DSA09-PRDC-010447 6.0x	ARM Limited	ARM Performance Monitoring Architecture version 2 Virtualization Extensions
6	ARM IHI 0014	ARM Limited	Embedded Trace Macrocell Architecture Specification
7	ARM IHI 0035	ARM Limited	CoreSight Program Flow Trace Architecture Specification

2 SCOPE

This document describes the Virtualization Extensions Architecture, an extension to the ARMv7-A profile and the Security Extensions.

3 VIRTUALIZATION EXTENSIONS OVERVIEW

3.1 Basic Model of Operation for Virtualization

The Virtualization Extensions provide hardware support for virtualization. The basic model for the virtualized system involves:

1. The hypervisor, which runs in a new Non-secure mode, called Hyp mode. The hypervisor is responsible for switching Guest Operating Systems (Guest OS)
2. A number of Guest OSes, which run in the Non-secure privileged and non-privileged modes

This model is based on providing virtualization support for Guest OSes that do not make use of the ARM Security Extensions other than, optionally, making calls to the secure side.

Secure software developed for use with the ARM Security Extensions is unchanged by this model, as the hypervisor makes no use of the Secure state. Any calls from the Guest OS to the Secure side can either be routed directly or are routed through the hypervisor (the control for this is under control of the hypervisor).

No support for Virtualization of Secure state functionality is provided at this time in the Virtualization Extensions.

The Virtualization Extensions can only be implemented if the Security Extensions have been implemented.

3.2 Overview of the Virtualization Extensions

The Virtualization Extensions provide facilities to aid the development of virtualized systems within the model described in section 3.1. The key aspects of the extensions are:

1. The provision of Hyp mode in the Non-secure state, and associated access mechanisms.
2. The provision of mechanisms to aid interrupt handling
3. The provision of mechanisms to aid memory management
4. The provision of mechanisms to aid the emulation of loads and stores to improve handling of Virtual Devices
5. The provision of mechanisms to trap particular situations to the hypervisor
6. Debug functionality.

These are described in turn in the following sections.

As these extensions build on the Security Extensions, they are only offered as extensions to the ARMv7-A profile.

4 HYP MODE

4.1 Introduction

The Virtualization Extensions introduces a new mode, Hyp mode, which can be entered when in Non-secure state. This new mode is more privileged than the other modes that can be accessed when in Non-secure state, in that it can control more functionality within the Non-secure state than the other modes.

Hyp mode is defined to be distinct from the rest of the Non-secure Privileged modes, described in [1, section B1.3.1]. and is not included in that set. In general, all registers and instructions that can be accessed in Privileged mode in Non-secure state can also be accessed from Hyp mode, unless otherwise stated in the specification.

The purpose of Hyp mode is to introduce a mode for use by the hypervisor.

4.1.1 Terminology

As the term “Privileged Mode” is potentially confusing when describing modes which are not the most privileged modes within a security state, a new term is introduced as part of the Virtualization Extensions. This term is the *Kernel Modes,* which is used to describe the following set of modes:

Supervisor, System, Abort, Undef, IRQ and FIQ

“Kernel Modes” is used exclusively within this document to describe modes in the Non-secure state.

The term *Privileged Modes* is retained to describe the following set of modes:

Supervisor, System, Abort, Undef, IRQ, FIQ and Monitor

“Privileged Modes” is used exclusively within this document to describe modes in the Secure state.

The term *Non-Privileged Modes* is retained to describe the following set of modes:

User

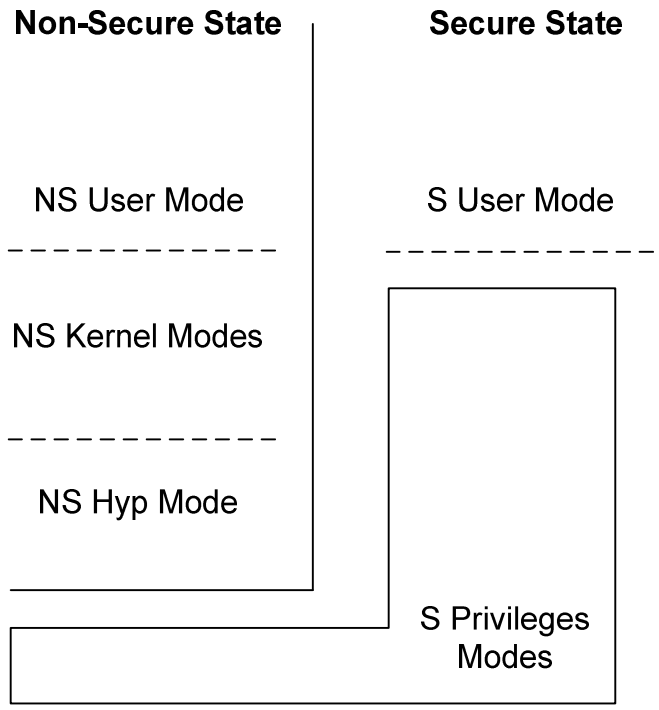
“Non-Privileged Modes” can be used in this document to describe modes in either the Secure state or the Non-secure state.

4.1.2 Hyp Mode Encoding

Hyp mode is given the Mode Encoding in the CPSR and SPSR of ‘11010’.

4.1.3 Diagrammatic representation of Hyp mode within the privilege hierarchy

The following diagram is designed to give a simple diagrammatic representation of the relationship between the different modes with the introduction of Hyp mode in the Non-secure state.



4.2 Additional General Purpose Registers

Hyp mode is a full mode, and so requires banked copies of SP (to provide a Stack pointer for use by the hypervisor at the point of exception entry to the hypervisor), and SPSR (to allow the previous mode of operation, interrupt masks etc to be captured).

The standard ARM approach for exception handling would be to bank the Link register (LR) in Hyp mode. However, this introduces a problem for the design of a hypervisor in that it prohibits the taking of exceptions within the hypervisor at times when its LR contains a procedure return. In a general virtualization solution, it is desirable that the hypervisor can enable interrupts (destined for the hypervisor) while running in the hypervisor mode, and to be able to do so without limiting the ability to use procedure calls.

For this reason, an alternative approach to handling the exception link information is adopted for the Hyp mode. This approach decouples the procedure link information from the exception link information. This involves:

- When executing in Hyp mode, LR is not "banked", but instead accesses LR_usr
- On taking an Exception whose destination is Hyp mode, the return link is placed into a new register, ELR_Hyp. The return link address holds the "preferred exception return" for the exception (as described in the ARM ARM section B1.6) without any offset; this avoids the need for software to perform arithmetic on the exception return address as part of the exception entry
- The ELR_Hyp is not a General Purpose register – it can only be explicitly accessed by the following mechanisms:
    - An MRS or MSR instruction targeting that register
    - An ERET instruction (see below)

ELR_Hyp and SPSR_Hyp become UNKNOWN as a result of execution in other Non-Secure modes.

4.2.1 ERET

A new instruction, ERET, is introduced, whose functionality, when executed in Hyp mode

SPSR_Hyp is transferred to CPSR

ELR_Hyp is transferred to PC

When executed in Secure Privileged modes and in Non-Secure Kernel modes, it behaves as MOVS PC, LR

ERET is UNPREDICTABLE in Debug state and in User mode.

Note:

The ERET instruction is the logical successor to MOVS PC, LR in the traditional ARM exception mechanism, but based on the ELR as holding the return address.

4.3 Additional restrictions associated with Hyp Mode

4.3.1 Accessing Hyp Mode from outside Hyp Mode

The following instructions are UNPREDICTABLE when executed in Non-Secure modes other than Hyp mode or when executed from the Secure state.

- MSR attempting to change into Hyp mode, except in Debug state when executed from the Non-secure state.
- CPS attempting to change into Hyp mode
- SRS attempting to use the Hyp mode SP

Note: Entry into Hyp mode or the use of Hyp mode resources would not be consistent with the definition of UNPREDICTABLE behaviour when these instructions are executed in Non-Secure modes other than Hyp mode.

4.3.2 Modifying CPSR.M when in Hyp mode

When in Hyp Mode:

- An MSR instruction which attempts to modify the CSPR.M bits is UNPREDICTABLE, except in Debug state.
- A CPS instruction which attempts to modify the CPSR.M bits is UNPREDICTABLE.

4.3.3 CPACR and Hyp mode

The CPACR has no effect on Coprocessor instructions or Advanced SIMD instructions executed in Hyp mode. The HCPTR register, described in section 8.14 controls Coprocessor instructions and Advanced SIMD instructions while executing in Hyp mode, as well as providing an additional control when executing in other Non-secure modes.

4.3.4 Instructions which are UNDEFINED or UNPREDICTABLE in Hyp mode

The following instructions are UNDEFINED when executed in Hyp mode:

- SRS
- RFE
- All ARM flag setting data processing operations with PC as the target register as defined in [1, section B6.1.13]
- Thumb `SUBS PC, LR, #N`, where N is not zero

  Note: ERET in Thumb state is encoded as `SUBS PC, LR, #0`
- LDM (Exception Return) [See 1, section B6.1.2]
- LDM (User Registers) [See 1, section B6.1.3]
- STM (User Registers) [See 1, section B6.1.11]
- SWP/SWPB

The following instructions are UNPREDICTABLE when executed in Hyp mode:

- LDRT, LDRSHT, LDRHT, LDRSBT, LDRBT, STRT, STRHT, STRBT

4.3.5 Exception Return to Hyp Mode

Performing an exception return with the SCR.NS=0 and the restored CPSR specifying Hyp mode is UNPREDICTABLE.

Performing an exception return from a Non-secure Kernel mode with the restored CPSR specifying Hyp mode is UNPREDICTABLE.

4.3.6 Use of Hyp Mode in Secure State

Hyp mode is introduced as a new mode, and level of privilege, within the Non-secure state, and the facilities that it provides, such as traps and memory translation, apply to operation in the Non-secure state. Hyp mode is not introduced into the Secure state, and there is no architecturally defined mechanism to allow entry into Hyp mode while Secure state.

Attempting to set the CPSR mode field to be set to the Hyp mode encoding in the Secure state is UNPREDICTABLE.

4.4 Taking exceptions into Hyp mode

On taking an exception into Hyp mode, the CPSR is copied to the SPSR_Hyp, and the preferred exception return address is written into the ELR_Hyp.

All exceptions, other than Interrupts, that cause a transition from a mode other than Hyp mode into Hyp mode, use a single vector offset – 0x14.

The behaviour is as shown in the Pseudo-code in section 16.2.1

4.4.1 Taking exceptions while executing in Hyp mode

When executing in Hyp mode, if an exception is taken and the exception is not routed to Monitor mode (as a result of the SCR.{IRQ, FIQ or EA} bits, or the exception is an SMC), then the exception is taken into Hyp mode, using

the standard vector offsets associated with the exception. On taking such an exception, the CPSR is copied to the SPSR_Hyp, and the preferred exception return address is written into the ELR_Hyp.

4.5 System Control Resources

A number of CP15 system registers for controlling the behaviour when in (or entering) Hyp mode are introduced as part of the Virtualization Extensions, where these values could be simultaneously used by the Guest OS and the hypervisor. The new registers can be accessed from Hyp mode or from Monitor mode, when SCR.NS==1 in the standard mechanism introduced in the Security Extensions.

Accesses to these new registers are UNDEFINED from the Kernel Modes within the Non-secure state, or from the Secure Privileged modes when SCR.NS ==0

Many of the system registers used within Hyp mode are new registers introduced as part of the Virtualization Extensions. However:

Some of the Hyp mode system registers are shared with the Secure copies of banked registers, where the security switching software is able to switch these registers as part of the switch between the secure and Non-secure states. This is possible where there is no requirement for these registers to be live simultaneously. Such registers can be accessed from Secure Privileged modes when SCR.NS==0 using the encodings for the Secure copies of the banked registers. The new encodings for use in Hyp mode are UNDEFINED from the Kernel Modes within the Non-secure state, or from the Secure Privileged modes when SCR.NS ==0

Some of the Hyp mode system registers are shared with the Non-secure copies of banked registers. This is possible where there is no requirement for the version used by the hypervisor view to be live at the same time as the same registers is live in a Guest OS. Such registers can be accessed from Non-secure Kernel modes.

Note: the new registers which are introduced for CP15 system control from Hyp mode and which are accessible from Hyp mode and from Monitor mode when SCR.NS==1 are logically “banked” registers for which the Secure version has not been implemented. This is consistent with the general position of the Virtualization extensions in not supporting Hyp mode in the Secure state. For the purpose of future documentation, these registers are “Banked-NSHyp only”.

4.5.1 Memory Management of Hyp Mode

When executing in Hyp mode, memory accesses are subject to translation using a different translation system than other Non-secure accesses. Accesses in Hyp mode only have one stage of translation.

This translation system uses the new translation table format described in [2], and consists of the following new registers:

```
HTTBR      MRRC/MCRR  p15, 4, Rt1, Rt2, c2
HTCR       MRC/MCR    p15, 4, Rt, c2, 0, 2
HMAIR0     MRC/MCR    p15, 4, Rt, c10, c2, 0
HMAIR1     MRC/MCR    p15, 4, Rt, c10, c2, 1
HSCTLR     MRC/MCR    p15, 4, Rt, c1, c0, 0;
```

When executing in Hyp mode, some bits in the translation tables have reserved values:

- nG bit is SBZ,
- APTable[0] bit is SBZ
- AP[1] bit is SBO
- PXNTable bit is SBZ
- PXN is SBZ
- NSTable and NS are Ignored by hardware.

The data fault address register and instruction fault address registers are shared with the Secure copy of the equivalent banked registers. The encodings below show the alias that can be used from Hyp mode to access this register.

```
HDFAR      MRC/MCR    p15, 4, Rt, c6, c0, 0     alias of the Secure DFAR
HIFAR      MRC/MCR    p15, 4, Rt, c6, c0, 2     alias of the Secure IFAR
```

For some second stage aborts that are taken in Hyp mode, the faulting IPA is presented in an additional register (the HPFAR); details of this are described in sections 6.2.7 and 13.6. This register is a 32-bit register which is new for the Virtualization Extensions.

```
HPFAR      MRC/MCR p15, 4, Rt, c6, c0, 4
```

4.5.2 Hyp Vector Base Address

The Vector Base Address for exceptions taken in Hyp Mode is a new register which is accessible from Hyp mode and from Secure Monitor (with SCR.NS==1) only.

```
HVBAR      MRC/MCR    p15, 4, Rt, c12, c0, 0
```

Note: In theory this register could be shared with the Secure VBAR, but that presents the security vulnerability that if an abort is taken in Monitor mode for some reason before the Secure VBAR has been reinstated, the Virtual Address used is determined from Non-secure state

4.5.3 Registers Shared between Hyp Mode and Non-secure Kernel modes

The following registers are shared between Hyp Mode and the Non-secure Kernel modes:

```
PAR:            Physical Address Register
CSSELR:         Cache Size Selection Register
ACTLR:          Auxiliary Control Register
```

4.6 Accessing Non-secure General Registers while in Hyp mode

The Security Extensions provide the same set of modes when running in the Secure state as in Non-secure state, and for the Security Extensions this provided a relatively straightforward way by which the banked general purpose registers could be accessed.

With the Virtualization Extensions, the hypervisor, running in Hyp mode, needs to be able to access all of the banked registers that are available to each Guest OS for the purpose of switching those registers between Guest OSes. This is performed by direct access to the banked registers using special MSR/MRS instructions.

This approach is generalised to all modes and provides simpler handling of a lot of context switch type operations (OS context switches, virtual machine switches and security switches), and so a full set of instructions is provided to access both banked general purpose registers and the ELR and SPSR special purpose registers:

```
MSR   <Rm>_<mode>, <Rn>
MSR   SPSR_<mode>, <Rn>
MSR   ELR_<mode>, <Rn>
MRS <Rn>, <Rm>_<mode>
MRS <Rn>, SPSR_<mode>
MRS <Rn>, ELR_<mode>
```

The instructions are available in all Non-secure Kernel modes, Hyp mode, and all Secure Privileged modes. They are UNPREDICTABLE in Secure User mode and Non-secure User mode.

The non-banked registers (R0-R7) cannot be accessed using this mechanism.

The registers that can be accessed are determined by the current mode of operation:

When in Monitor mode, the following registers cannot be selected using this mechanism:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_mon, LR_mon, SPSR_mon

When in Hyp mode:

If NSACR.RFR is clear, then the following registers cannot be selected using this mechanism:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, LR_usr, SP_hyp, SPSR_hyp, SP_mon, LR_mon, SPSR_mon.

If NSACR.RFR is set, then the following registers cannot be selected using this mechanism:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_hyp, SPSR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq.

When in Kernel modes, when SCR.NS is set :

If NSACR.RFR is clear, then the following registers cannot be selected using this mechanism:

If in FIQ mode:

R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon.

If not in FIQ mode:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_<current_mode>, LR_<current_mode>, SPSR_<current_mode>, SP_hyp,SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon

If NSACR.RFR is set, then the following registers cannot be selected using this mechanism:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_<current_mode>, LR_<current_mode>, SPSR_<current_mode>, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq,

When in Privileged modes other than Monitor mode, when SCR.NS is clear, then the following registers cannot be selected using this mechanism:

If in FIQ mode:

R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq, SP_hyp, SPSR_hyp, ELR_hyp,

If not in FIQ mode:

R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_<current_mode>, LR_<current_mode>, SPSR_<current_mode>, SP_hyp, SPSR_hyp, ELR_hyp,

If a register cannot be selected, then performing these instructions with that register is UNPREDICTABLE.

Note: Providing access to registers that are associated with a mode that cannot be entered using a CPS or MSR instruction is not consistent with the definition of UNPREDICTABLE.

Note: The behaviour of the MSR/MRS instructions of the modes contains two elements:

1. Not being able to access registers that, for reasons of privilege and security, are not accessible to the current mode of execution. This is required to prevent significant security holes.
2. Not being able to access the registers associated with the current mode that are accessible by other means, such as regular MOV instructions or alternative MSR/MRS instructions. Making these cases UNPREDICTABLE permits implementation freedom regarding the mechanisms for holding the registers associated with other modes.

Note: These instructions use the instruction format described in section 14.3. The existing MSR/MRS instructions for access the SPSR described in [1] are unaffected by these new instructions.

In Debug State, these instructions behave the same way as in Non-debug state.

For each of use, these restrictions are shown in a tabular form below. **Note:** this table should not contradict any information presented above.

Current Mode	Secure state	Non-Secure state if NSACR.RFR==0	Non-Secure state if NSACR.RFR==1
MON	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_mon, LR_mon, SPSR_mon	N/A	N/A
HYP	N/A	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, LR_usr, SP_hyp, SPSR_hyp, SP_mon, LR_mon, SPSR_mon	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, LR_usr, SP_hyp, SPSR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
FIQ	R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq, SP_hyp, SPSR_hyp, ELR_hyp	R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon	N/A
IRQ	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_irq, LR_irq, SPSR_irq, SP_hyp, SPSR_hyp, ELR_hyp	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_irq, LR_irq, SPSR_irq, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon,	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_irq, LR_irq, SPSR_irq, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
UND	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_und, LR_und, SPSR_und, R13_hyp, SPSR_hyp, ELR_hyp	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_und, LR_und, SPSR_und, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon,	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_und, LR_und, SPSR_und, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
ABT	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_abt, LR_abt, SPSR_abt, SP_hyp, SPSR_hyp, ELR_hyp	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_abt, LR_abt, SPSR_abt, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon,	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_abt, LR_abt, SPSR_abt, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
SVC	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_svc, LR_svc, SPSR_svc, SP_hyp, SPSR_hyp, ELR_hyp	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_svc, LR_svc, SPSR_svc, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon,	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_svc, LR_svc, SPSR_svc, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
SYS	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_usr, LR_usr, SP_hyp, SPSR_hyp, ELR_hyp	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_usr, LR_usr, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon,	R8_usr, R9_usr, R10_usr, R11_usr, R12_usr, SP_usr, LR_usr, SP_hyp, SPSR_hyp, ELR_hyp, SP_mon, LR_mon, SPSR_mon, R8_fiq, R9_fiq, R10_fiq, R11_fiq, R12_fiq, SP_fiq, LR_fiq, SPSR_fiq
USR	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE

4.7 System Call into Hyp Mode

A new instruction, HVC, is added to allow calling of the Hyp mode from within the Non-secure Kernel modes. The instruction is UNDEFINED when:

- In Secure state
- In Non-secure User mode
- The SCR.HCE bit is 0.

This instruction (if it is not UNDEFINED) causes an exception which enters Hyp mode. The vector used depends on whether the mode that the instruction was executed in was a Non-Secure Kernel mode, or was Hyp mode:

- If the HVC was executed in Hyp mode, the vector offset is 0x8
- If the HVC was executed in a Non-Secure Kernel mode, the vector offset is 0x14

The preferred exception return from an HVC exception is a return to the next instruction after the HVC instruction.

The HVC instruction is UNPREDICTABLE in Debug State.

HVC is an unconditional instruction.

The immediate argument of the HVC instruction is captured in the Hyp Syndrome Register (HSR) – see sections 7.1 and 13.7

4.7.1 SVC executed in Hyp Mode

If SVC is executed in Hyp mode, it behaves in the same way as HVC executed in Hyp mode, causing an exception into Hyp mode with the vector offset of 0x8. The SVC is distinguished from an HVC in the HSR.

If SVC is unconditional, the immediate field in the HSR in this case takes:

A zero-extended 8bit immediate value for the 16-bit Thumb SVC instruction

The bottom16 bits of the immediate for the 32-bit ARM SVC instruction

If SVC is conditional, the immediate field in the HSR is UNKNOWN.

5 ASYNCHRONOUS EXCEPTION CONTROLS

5.1 Asynchronous Exception Routing and Control

5.1.1 Overview

Interrupts and asynchronous data aborts present a particular problem for virtualization as it is necessary to route the exception to the correct recipient. In general, there are a number of possible recipients:

- The secure monitor
- The hypervisor
- The currently running Guest OS
- A Guest OS that is not currently running

In addition, it is often necessary for a Guest OS to prevent interrupts that will be handled by that Guest OS from being taken during critical sections by the use of the CPSR Interrupt Masks. However, it is required for full virtualization that a Guest OS cannot mask interrupts that need to be routed to either the hypervisor or a different Guest OS.

The basic model assumed in the Virtualization Extensions is that Non-secure interrupts and, in some cases (according to the system requirements), asynchronous data aborts that are known not to be destined to the currently running Guest OS are routed to the hypervisor. Secure interrupts and, in some cases if so configured, asynchronous data aborts are routed to the Secure monitor.

5.1.2 Terminology

The Virtualization Extensions introduce a distinction between *Physical* Asynchronous exceptions and *Virtual* Asynchronous exceptions. The Asynchronous exceptions FIQ, IRQ and Asynchronous Abort that were defined in [1] are described in the Virtualization Extensions as being Physical Asynchronous Exceptions. The Virtualization Extensions introduce an equivalent set of Virtual Asynchronous Exceptions:

- Virtual IRQ
- Virtual FIQ
- Virtual Asynchronous Abort

The Virtual Asynchronous Exceptions can be used to allow an Asynchronous Exception to be recorded for a Virtual Machine by a Hypervisor.

5.1.3 Routing Control

Routing of physical interrupts/aborts is determined by a combination of the routing controls in the SCR, (the SCR.{IRQ,FIQ, EA} as defined by the Security Extensions), and a set of interrupt controls, known as the Mask Override bits, in the Hyp Configuration Register (HCR) introduced as part of the Virtualization Extensions. The SCR control has priority over the HCR control (so allowing the Secure code to have the primary control over the routing of interrupts).

The Mask Override bits in the HCR have the following effects when set:

- They determine that the virtual interrupt/abort bits in the HCR have an effect. This control is orthogonal to the SCR control of the routing of the interrupts/abort to the Secure side, and so introduces the facility of having Virtual Interrupts/Aborts of different types set by the hypervisor independently of the handling of the physical interrupts/aborts.
- They determine that the CPSR Interrupt and Asynchronous Abort Masks do not mask Physical Interrupts or Asynchronous Aborts for the following cases:
    - For all Non-secure modes if the corresponding physical interrupt or abort is routed to Monitor mode using the SCR controls.
    - For all Non-secure modes other than Hyp mode if the corresponding physical interrupt or abort is not routed to Monitor mode using the SCR controls.

  In all other cases, the CPSR Interrupt and Asynchronous Abort Masks mask Physical Interrupts or Asynchronous Aborts
- They determine that the CPSR Interrupt and Asynchronous Abort Masks mask Virtual Interrupts or Asynchronous Aborts when executing in all Non-secure modes other than Hyp mode

The Security Extensions include control bits in the SCR, the SCR.FW and SCR.AW bits, that restrict the ability for the CPSR.F and CPSR.A bits to be modified from within the Non-secure state. The purpose of this functionality is to prevent code running in Non-secure state from being able to mask interrupts/aborts that are to be handled by the Secure side; this ensures delivery of those exceptions to the Secure code. However, the combination of these controls and the associated mask override and the routing controls lead to situations where the Virtual Interrupt/Aborts are not useful. To avoid this, and to simplify the logic, the SCR.FW and SCR.AW bits are redefined by the Virtualization Extensions, so that when clear they cause the CPSR.F and CPSR.A bits to have no effect in masking interrupts or asynchronous external aborts when:

- Executing in any Non-Secure mode, and
- the associated mask override bit is not set, and
- the associated exception is routed to Monitor mode

In all other cases, the SCR.FW and SCR.AW bits have no effect

The following tables show the relationship between the various controls for interrupt routing:

IRQ

SCR.IRQ	**HCR.IMO**	**Behaviour**

0	**0**	Physical IRQ taken in IRQ mode unless executing in Hyp mode; when executing in Hyp mode, the Physical IRQ is taken in Hyp mode. Virtual IRQ bit has no effect CPSR.I bit can mask Physical IRQ in all states and modes
0	**1**	Physical IRQ taken in Hyp mode when in Non-secure state; Physical IRQ taken in IRQ mode when in Secure state Virtual IRQ bit used to signal Virtual Interrupts in the Non-secure state other than Hyp mode; these are taken in IRQ mode. Virtual IRQ bit has no effect when in Secure state or in Hyp mode. CPSR.I bit can not mask Physical IRQ when in Non-secure state other than Hyp mode, but can mask Virtual IRQ. CPSR.I bit can mask Physical IRQ in Secure state or in Hyp mode
1	**0**	Physical IRQ taken in Monitor mode Virtual IRQ bit has no effect CPSR.I bit can mask Physical IRQ in all states and modes
1	**1**	Physical IRQ taken in Monitor mode Virtual IRQ bit used to signal Virtual Interrupts in the Non-secure state other than Hyp mode; these are taken in IRQ mode. Virtual IRQ bit has no effect when in Secure state or in Hyp mode CPSR.I bit cannot mask Physical IRQ when in Non-secure state. CPSR.I bit can mask Physical IRQ when in Secure state When in Non-secure state other than Hyp mode, the CPSR.I bit can mask Virtual IRQ.

Asynchronous Abort

SCR.EA	SCR.AW	HCR.AMO	Behaviour
0	**X**	**0**	Physical Asynchronous Abort taken in Abort mode unless executing in Hyp mode; when executing in Hyp mode, the Physical Asynchronous Abort is taken in Hyp mode. Virtual Asynchronous Abort bit has no effect CPSR.A bit can mask Physical Asynchronous Aborts in all states CPSR.A bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes. See **Note** below
0	**X**	**1**	Physical Asynchronous Abort taken in Hyp mode when executing in Non-secure state, and in Abort mode when executing in Secure state Virtual Asynchronous Abort bit used to signal Virtual Asynchronous Aborts in the Non-secure state other than Hyp mode; these are taken in Abort mode. Virtual Asynchronous Abort bit has no effect when in Secure state or in Hyp mode CPSR.A bit cannot mask Physical Asynchronous Aborts when in Non-secure state other the Hyp mode, but can mask Virtual Asynchronous Aborts. CPSR.A bit can mask Physical Asynchronous Aborts when in Secure state or in Hyp mode.

			CPSR.A bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.
1	**0**	**0**	Physical Asynchronous Abort taken in Monitor mode Virtual Asynchronous Abort bit has no effect CPSR.A bit cannot mask Physical Asynchronous Aborts when in Non-secure state CPSR.A bit can mask Physical Asynchronous Aborts when in Secure state CPSR.A bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.
1	**1**	**0**	Physical Asynchronous Abort taken in Monitor mode Virtual Asynchronous Abort bit has no effect CPSR.A bit can mask Physical Asynchronous Aborts in all states CPSR.A bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.
1	**X**	**1**	Physical Asynchronous Abort taken in Monitor mode Virtual Asynchronous Abort bit used to signal Virtual Asynchronous Aborts in the Non-secure state other than Hyp mode; these are taken in Abort mode. Virtual Asynchronous Abort bit has no effect when in Secure state or Hyp mode CPSR.A bit cannot mask Physical Asynchronous Aborts when in Non-secure state. CPSR.A bit can mask Physical Asynchronous Aborts when in Secure state. When in Non-secure state other than Hyp mode, the CPSR.A bit can mask Virtual Asynchronous Aborts. CPSR.A bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.

Note: The behaviour in Non-secure state when SCR.EA==0 and SCR.AW==0 is changed in the Virtualization Extensions to remove a behaviour that was not useful, as it can result in repeated entry into Abort mode, and which produced unnecessary complication with the addition of the HCR.AMO bit.

FIQ

SCR.FIQ	SCR.FW	HCR.FMO	Behaviour
0	**X**	**0**	Physical FIQ taken in FIQ mode unless executing in Hyp mode; when executing in Hyp mode, the Physical FIQ is taken in Hyp mode. Virtual FIQ bit has no effect CPSR.F bit can mask Physical FIQ in all states CPSR.F bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes See **Note** below
0	**X**	**1**	Physical FIQ taken in Hyp mode when executing in Non-secure state, and in FIQ mode when executing in Secure state Virtual FIQ bit used to signal Virtual FIQ interrupts in the Non-secure state

			other than Hyp mode; these are taken in FIQ mode. Virtual FIQ bit has no effect when in Secure state or in Hyp mode. CPSR.F bit cannot mask Physical FIQ when in Non-secure state other than Hyp mode, but can mask Virtual FIQ. CPSR.F bit can mask Physical FIQ when in Secure state and when in Hyp mode. CPSR.F bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes
1	**0**	**0**	Physical FIQ taken in Monitor mode Virtual FIQ bit has no effect CPSR.F bit cannot mask Physical FIQ when in Non-secure state CPSR.F bit can mask Physical FIQ when in Secure state CPSR.F bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.
1	**1**	**0**	Physical FIQ taken in Monitor mode Virtual FIQ bit has no effect CPSR.F bit can mask Physical FIQ in all states CPSR.F bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.
1	**X**	**1**	Physical FIQ taken in Monitor mode Virtual FIQ bit used to signal Virtual FIQ interrupts in the Non-secure state other than Hyp mode; these are taken in FIQ mode. Virtual FIQ bit has no effect when in Secure state or Hyp mode CPSR.F bit cannot mask on Physical FIQ when in Non-secure state. CPSR.F bit can mask Physical FIQ when in Secure state When in Non-secure state other than Hyp mode, the CPSR.F bit can mask Virtual FIQ. CPSR.F bit can be modified in Non-secure state in Hyp and Kernel modes and in Secure state Privileged modes.

Note: The behaviour in Non-secure state when SCR.FIQ==0 and SCR.FW==0 is changed in the Virtualization Extensions to remove a behaviour that was not useful, as it can result in repeated entry into FIQ mode, and which produced unnecessary complication with the addition of the HCR.FMO bit.

In all cases:

If NSACR.RFR==1 then HCR.FMO behaves as if it is 0, regardless of the physical setting of the HCR.FMO bit.

If either HCR.FMO==1 or HCR.IMO==1 then the Non-secure SCTLR.VE bit behaves as if it is 0, regardless of the physical setting of the SCTLR.VE bit.

5.2 Virtual Interrupt and Abort bits

The Virtualization Extensions introduce 3 virtual interrupt bits, used to hold the pending interrupt or asynchronous abort state for the current Guest OS. There is one bit for each of Asynchronous Aborts, IRQ interrupts and FIQ interrupts. These bits are held in the Hyp Configuration Register (HCR) which is a new register introduced as part of the Virtualization Extensions.

The virtual interrupts or aborts are taken within the Guest OS in the same way as the corresponding Physical interrupt or abort would be taken if the SCR {EA, IRQ, FIQ} bits and HCR.{AMO, IMO, FMO} bits are not set. A virtual interrupt or abort is only taken when the corresponding CPSR mask bit is clear.

When a virtual abort is taken, the HCR.VA bit is cleared automatically by hardware. When a virtual interrupt is taken, the corresponding bit in the HCR (the HCR.VI or HCR.VF bit) is not changed by hardware (and so must be cleared as part of the interrupt service routine).

The effect of virtual exceptions on WFI and WFE wake-up events is discussed in section 5.3

Version 2 of the Generic Interrupt Controller (GICv2) adds support for virtual interrupts that complement the provisions in this specification. The changes are documented in the next release of the GIC specification [3].

5.2.1 Interrupt Status Register

The Interrupt Status Register displays the pending nature of interrupts (and Asynchronous Aborts). Under the Virtualization Extensions, the values returned by a read of the ISR become a function of the mode and security state of the read, as follows:

- When accessed in Secure state, they show the status of the Physical Interrupts/Asynchronous Abort (regardless of the value of the SCR.NS bit)
- When accessed from Hyp mode in the Non-secure state, they show the status of the Physical Interrupts/Asynchronous Abort
- When accessed from the Kernel modes in the Non-secure state, then for each interrupt/abort, the status shown is a function of the corresponding Mask Override control:
    - If the Mask Override is clear, the corresponding ISR bit shows the status of the Physical Interrupt/Asynchronous Abort
    - If the Mask Override is set, the corresponding ISR bit shows the status of the Virtual Interrupt/Asynchronous Abort

5.3 WFE and WFI Wake-up Events

The ARMv7 definitions of WFE and WFI wake-up events in [1, sections B1.6.8 and B1.6.9] are modified by the Virtualization extensions in the following ways:

- When in Non-secure modes other than Hyp mode, Virtual interrupts and aborts can act as wakeup events
- The strict definition of WFE Wake-up events is modified in the light of the changes to interrupt masking.

As result, the relevant WFI and WFE Wake-up events are:

WFI Wakeup events:

- A physical IRQ interrupt, regardless of the value of the CPSR.I bit
- A physical FIQ interrupt, regardless of the value of the CPSR.F bit
- A physical asynchronous abort, regardless of the value of the CPSR.A bit
- A virtual IRQ interrupt, when in a Non-secure mode other than Hyp, and HCR.IMO is set, regardless of the value of the CPSR.I bit
- A virtual FIQ interrupt, when in a Non-secure mode other than Hyp, and HCR.FMO is set, regardless of the value of the CPSR.F bit
- A virtual asynchronous abort, when in a Non-secure mode other than Hyp, and HCR.AMO is set, regardless of the value of the CPSR.A bit
- A debug event, when invasive debug is enabled and the debug event is permitted.

WFE Wakeup events:

- The execution of an SEV instruction on any processor in the multiprocessor system
- A physical IRQ interrupt that is not masked by the CPSR.I bit

- A physical FIQ interrupt that is not masked by the CPSR.F bit
- A physical asynchronous abort that is not masked by the CPSR.A bit
- A virtual IRQ interrupt that is not masked by the CPSR.I bit when in a Non-secure mode other than Hyp, and HCR.IMO is set
- A virtual FIQ interrupt that is not masked by the CPSR.F bit, when in a Non-secure mode other than Hyp, and HCR.FMO is set
- A virtual asynchronous abort that is not masked by the CPSR.A bit, when in a Non-secure mode other than Hyp, and HCR.AMO is set
- A debug event, when invasive debug is enabled and the debug event is permitted.

5.4 Low Interrupt Latency Configuration

In the ARMv7 architecture, the SCTLR.FI bit is common between the Secure and Non-secure states, and can only be set from the secure state. Implementations are permitted not to implement the SCTLR.FI if they do not provide low interrupt latency facilities. The SCTLR.FI bit is unchanged by the Virtualization Extensions.

5.5 Non-Maskable Fast Interrupt

The Virtualization Extensions do not support Non-Maskable Fast Interrupt, and so all implementations that include the virtualization extensions, the SCTLR.NMFI bit is 0.

5.6 Effects on the Interrupt Controller

The virtual cpu interface supported by the GICv2 architecture provides a means for a hypervisor to assign and manage lists of interrupts for the currently executing virtual machine. This will be documented in the next release of [3].

5.7 Effects on debug

The debug Interrupts Disable bit (see [1]), DBGDSCR.INTdis, masks the taking of all IRQ and FIQ exceptions, including those due to Virtual Interrupts. The INTdis bit is ignored when either DBGDSCR[15:14] == 0b00 or DBGEN is LOW. INTdis does not affect the value returned by reads of the ISR.

6 MEMORY MANAGEMENT

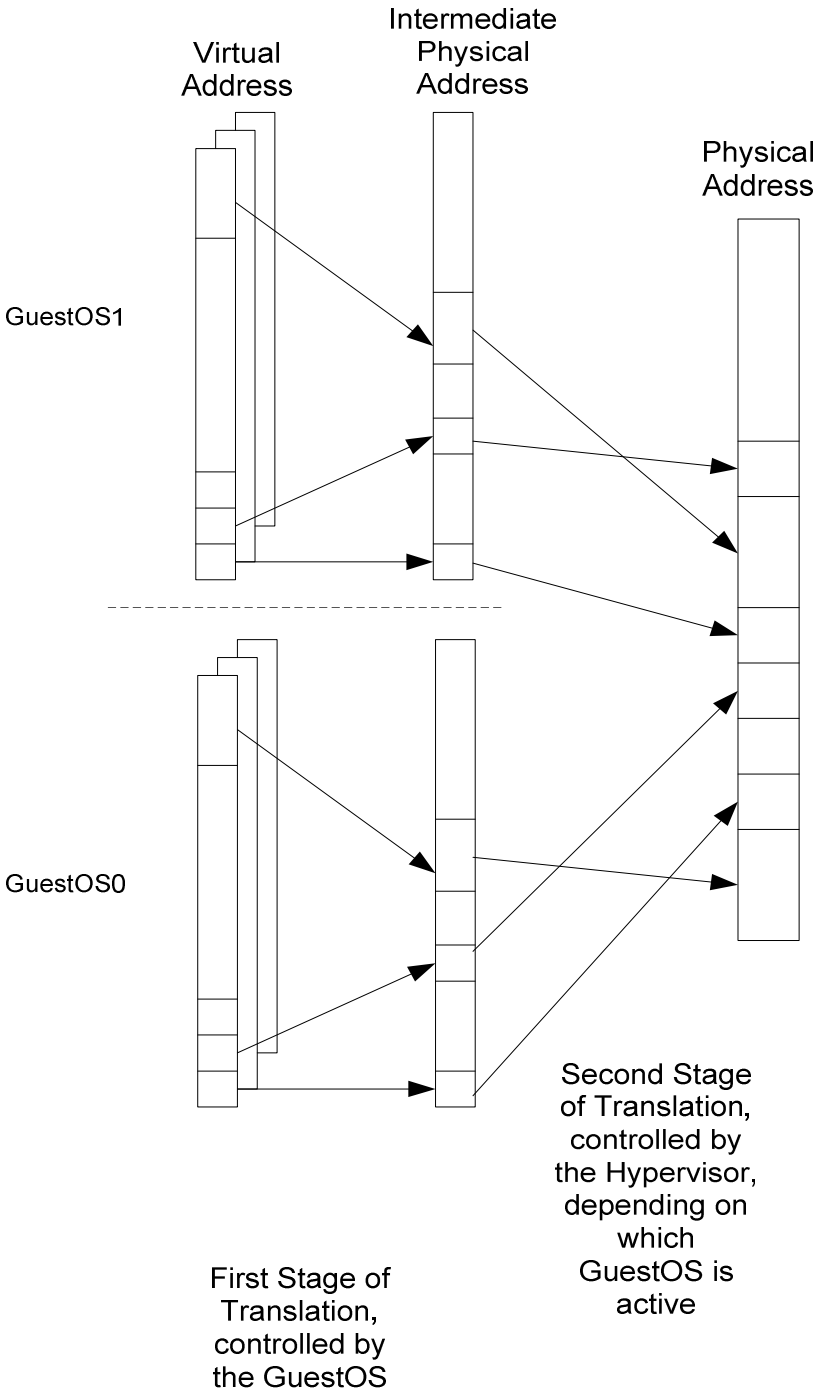
6.1 Overview

One of the key functions of many operating systems is the management of the memory of its applications, and this code can have substantial complexity. In particular, most platform operating systems support a stage of virtual memory management to partition the memory controlled by the operating system across the various applications.

In a system where each Guest OS is running in a Virtual Machine, the memory that is being allocated by the Guest OS is not the true physical memory of the system, but instead is *intermediate physical memory* provided by the hypervisor. This then allows the hypervisor to control the allocation of the actual physical memory to the individual Guest OSes according to the needs of the whole system. Therefore in such a system, there are three levels of address space provided:

- Virtual Address (VA), being the address space that an application uses
- The Intermediate Physical Address (IPA), being the address space that the Virtual Address is mapped to by the Guest OS
- The Physical Address (PA), being the address space that the Intermediate Physical Address is mapped to by the hypervisor.

The following diagram shows the relationship:



A key consideration in the virtualized system is the complexity of the relationship between the intermediate physical memory of each guest OS and the true physical memory of the system. There are a range of plausible relationships that could be envisaged, for example

- The intermediate physical address is actually the same as the actual physical address, and so the hypervisor performs no additional address translation. In general, this requires each Guest OS to be built to access a different intermediate physical memory space, and the role of the hypervisor is to ensure that each Guest OS does not stray beyond its legal intermediate physical memory space.

- The intermediate physical address is offset by a different constant for each Guest OS to create the actual physical address. In this case, the memory used by each Guest OS is contiguous in the physical address map.
- Each block of physical memory could be mapped independently to one or more Guest OSes. A variety of sizes of the blocks could be envisaged, though a likely approach would be to allocate blocks on the granularity of the page or section sizes of the underlying hardware architecture. This is referred to as a fragmented relationship.

As the number of Guest OSes increase, the relationship between the intermediate physical memory of a Guest OS and the physical memory is likely to become more complex. In particular, two kinds of systems seem to create a need to have a fragmented relationship between these two levels of memory:

- Systems which dynamically create and destroy different Guest OSes while the system is running
- Systems in which regions of physical memory are shared between different Guest OSes.

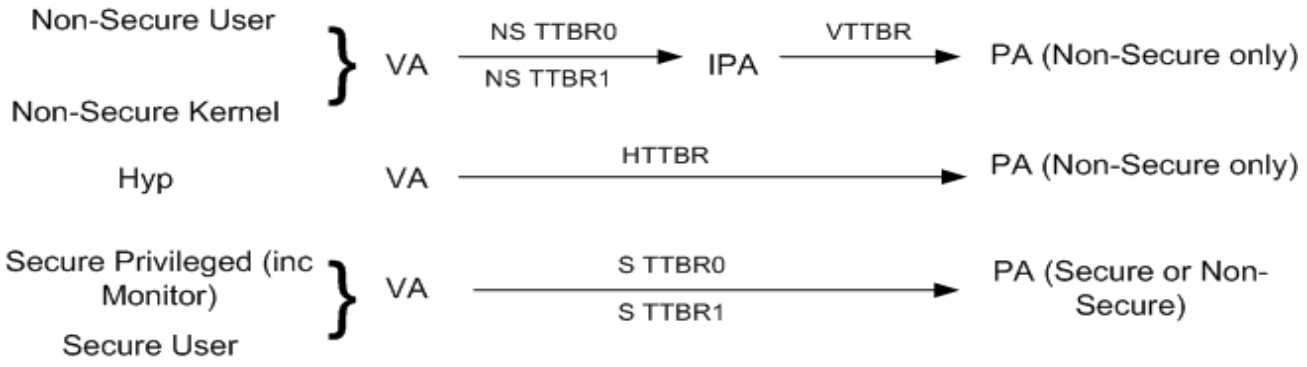
For these reasons, the Virtualization Extensions allow for a fragmented relationship between the intermediate physical address and the physical address.

6.2 Details of a two stage hardware translation

6.2.1 Overview of the Architectural Support for two stages of address translation

Under the Virtualization Extensions, all “Physical” addresses generated while executing from the Kernel and Non-Privileged modes in the Non-secure state are renamed as Intermediate Physical Addresses (IPAs). All IPAs can be made subject to a second stage of translation to become a PA.

Translations of Virtual addresses while executing from Hyp mode or from the Secure state use separate single (first) stages of translation and are unaffected by the second stage of translation. This is shown in the diagram below:



This second stage of translation is based on a multiple level translation table scheme similar in concept to the existing ARMv7-A Translation Table scheme which is used for VA to IPA translation. However, it does not have all of the features (and in particular the legacy features) of the ARMv7-A Translation Table scheme. In addition, the second stage system is based on a translation scheme with a 64-bit descriptor to allow it to address a physical memory of larger than 4GBytes

The IPA to PA translation is explicitly enabled via a control bit within the Hyp Configuration Register (HCR). When the IPA to PA translation is disabled (as from reset), the translation is a direct translation from IPA to PA (with 0 extension of the upper bits of the PA if appropriate).

Faults generated by the second stage of translation are taken in Hyp mode, going to the vector offset used for entries into Hyp mode (0x14), so the vector address is therefore Hyp Vector Base Address + 0x14 in all cases. The HSR register holds syndrome information for such faults.

Faults generated by the first stage of translation are taken in the Non-secure abort mode using the Non-secure Prefetch or Data abort vectors as appropriate. In essence, the Virtualization Extensions creates two different sets of data or prefetch aborts, one for stage 1 translation (VA to IPA) and a second for stage 2 translation (IPA to PA).

The second stage of translation does not have all of the features of the first stage. In particular:

- All entries in the IPA to PA translation are associated with the current Guest OS ID (the VMID – see section 6.2.4); there is no equivalent of the global translation table entry.
- There is no concept of domains within the second stage of translation – all entries behave as if they have client access
- The second stage handling of attributes and permissions is simplified from the first stage handling – this is described below.
- The second stage handling does not have two translation table base registers.

An Access Fault in the second stage of translation is supported.

6.2.2 Access permission handling in a two stage memory translation system

6.2.2.1 Non-secure First Stage Permission for Kernel and Non-Privileged modes.

The Non-secure first stage translation applies only to accesses made from Kernel and Non-Privileged modes. The permission check for “privileged” access as part of the first stage of translation applies to memory accesses performed from the Kernel modes, other than those generated by LDRT, LDRSHT, LDRHT, LDRSBT, LDRBT, STRT, STRHT or STRBT instructions.

6.2.2.2 Second Stage Permission

The second stage of translation provides a level of control for access permissions that is set by the hypervisor, in addition to that being set by the Guest OS. For regions of memory shared between different Guest OSes, the hypervisor is able to control such access permission as an additional set of restrictions. As such, the final access permission becomes the intersection of the access permitted by the Guest OS and the access permitted by the hypervisor, so that permission to read or to write must be granted by both the permissions in the first stage of translation and the permissions in the second stage of translation.

There is no need for the hypervisor owned access permissions to set permissions based on User or Kernel level request – the hypervisor is not aware of this distinction. However, there can be advantages in allowing the hypervisor to assign a “Write-Only” permission to apply to a region in the Guest OS.

As a result, the second stage translation tables contain a 2 bit field (HAP[1:0]) to describe the access permissions.

HAP[1]	HAP[0]	Meaning
0	0	No Access
0	1	Read-only
1	0	Write-only
1	1	Read/Write access

6.2.3 Memory type handling

The second stage of translation provides a level of control for memory type that is set by the hypervisor, in addition to that being set by the Guest OS. In particular, it can be useful for the hypervisor to reduce the permitted cacheability options and to increase the shareability options for a region. To do this involves combining the cacheability and shareability attributes from the second stage of translation with the first stage translation, following a “lowest common denominator approach” – this is shown in more detail in 6.2.3.1

The memory type information for the second stage of translation is encoded in a single fixed field of 4 bits. A separate field encodes the Shareability attributes.

The second stage of translation doesn't provide the ability to assign allocation hints, unlike the first stage of translation.

Normal, Inner WB Cacheable, Outer WB Cacheable, Non-Shareable for the second stage of translation equates to "No change" to the attributes generated by the first stage of translation.

> **Note**: The second stage of translation does not have as much state associated with it as the first stage of translation and the indirection mechanism of the PRRR and NMRR used in the first stage of translation is not supported for the second stage tables.

These values are decoded based on a 4 bit field MemAttr as shown in the following tables

MemAttr[3:2]	**Meaning**
00	SO or Device Memory
01	Normal Memory, Outer Non-Cacheable
10	Normal Memory, Outer Write-through
11	Normal Memory, Outer Write-back

MemAttr[1:0]	**Meaning when Bits[3:2] == '00'**	**Meaning when Bits[3:2] != '00'**
00	Strongly Ordered	UNPREDICTABLE
01	Device	Normal Memory Inner Non-Cacheable
10	UNPREDICTABLE	Normal Memory Inner Write-through
11	UNPREDICTABLE	Normal Memory Inner Write-back

6.2.3.1 Tables for combining memory attributes

Setting in Stage 1	**Setting in Stage 2**	**Resultant Primary Type**
Strongly-Ordered	X	Strongly-Ordered
X	Strongly-Ordered	Strongly-Ordered
Device	Normal or Device	Device
Normal or Device	Device	Device
Normal	Normal	Normal

For Normal memory, the cacheability attributes follow the following rules:

Setting in Stage 1	**Setting in Stage 2**	**Resultant Cacheability**
Non-Cacheable	X	Non-Cacheable
X	Non-Cacheable	Non-Cacheable

Write-Through	Write-Through or Write-Back	Write-Through
Write-Through or Write-Back	Write-Through	Write-Through
Write-Back	Write-Back	Write-Back

This table applies independently for both Inner and Outer cacheability attributes.

For Normal memory, the shareability attributes follow the following rules:

Setting in Stage 1	**Setting in Stage 2**	**Resultant Shareability**
Outer Shareable	X	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-Shareable	Inner Shareable
Non-Shareable	Outer Shareable	Outer Shareable
Non-Shareable	Inner Shareable	Inner Shareable
Non-Shareable	Non-Shareable	Non-Shareable

For implementations that include the Virtualization Extensions:

- If the attributes for a memory location after combination of the 1st and 2nd stages are Strongly-ordered or Device, then they are treated as Outer Shareable (see also [2] for a discussion of the significance of Shareability for the Strongly-Ordered and Device memory types)
- If the attributes for a memory location after combination of the 1st and 2nd stages is Normal Inner Non-Cacheable, Outer Non-Cacheable, then the shareability attributes after combination of the 1st and 2nd stages must be Outer Shareable, or else the shareability is UNPREDICTABLE

First Stage Translation Table walks to Strongly-Ordered or Device Memory locations

A consequence of the attribute combining is that it becomes possible for translation table walks associated with the first stage of translation to be made to Strongly-ordered or Device memory. While the architecture prohibits speculative data accesses to Strongly-ordered or Device memory, it does not prevent translation table walks to be made speculatively to such memory types. The virtualization extensions provide a trap to allow these accesses to be faults, as described in section 6.2.9

6.2.4 Virtual Machine Identification

Architecturally, a new identification concept becomes visible – the Virtual Machine Identification Register (VMID). This serves a very similar purpose to the ASID, but at the next stage of address handling (that is, it identifies virtual machines, each of which has its own set of applications, and therefore its own ASID space). Each Guest OS is assigned a VMID by the hypervisor, and so the architecture does not require explicit invalidation by software of the TLB when changing between translations for the same ASID and VA, but with a different VMID.

6.2.5 Atomicity of register changes

In the ARMv7-A architecture, the code sequence for changing the ASID and the TTBRx registers is relatively complicated to avoid any possibility of the new ASID being associated with translations from the old mapping (or vice-versa). This is discussed in the [1, section B3.10.5]. There is an equivalent synchronization issue in a virtualization system, when switching between different virtual machines. There are two sets of registers which need to be changed atomically from the viewpoint of execution in a Non-secure mode other than Hyp mode if incorrect mappings are to be avoided; these sets are:

- The first stage translation table registers for use by a Non-secure mode other than Hyp mode.

- The second stage translation registers for use by a Non-secure mode other than Hyp mode.

For the first stage translation table registers, the set of registers that need to appear to be updated atomically from the viewpoint of execution in a Non-secure mode other than Hyp mode includes the PRRR/MAIR0, NMRR/MAIR1, TTBR0, TTBR1, TTBCR, DACR, CONTEXTIDR and some SCTLR register bits.

For the second stage translation table registers, the set of registers that need to appear to be updated atomically from the viewpoint of execution in a Non-secure mode other than Hyp mode includes the PRRR/MAIR0, NMRR/MAIR1, TTBR0, TTBR1, TTBCR, DACR, CONTEXTIDR and some SCTLR register bits.

The switching of these registers, as part of a switch of virtual machines, occurs while executing in Hyp mode, but the registers apply to execution when in a Non-secure mode other than Hyp mode, and as such the registers being switched are out of context.

The Virtualization Extensions introduce a constraint that it is not legal for an implementation to use the memory translation registers used by the Non-secure modes other than Hyp mode for speculative memory accesses when executing in Hyp mode or in Secure state.

6.2.6 Instruction and Data/Unified Cache considerations from Virtualization

Data/Unified Caches

In keeping with the ARMv7-A architecture [1, section B3.4], the behaviour of any data or unified caches must be consistent with their implementation as PIPT caches. In particular, accesses to a cacheable or non-cacheable physical address from the same processor (or from other processors within the same shareability domain) using different VA or IPA addresses must behave in a coherent manner without the need for explicit cache maintenance.

Instruction Caches

In keeping with the ARMv7-A architecture [1, section B3.4.1], the behaviour of the instruction cache varies depending on whether the IVIPT extensions are implemented. If the IVIPT extensions are not implemented, the introduction of the second stage of translation adds additional cases in which instruction cache maintenance must be performed:

- Changing one or more of the VTTBR.BADDR field and the VTCR registers without changing the VMID
- Enabling or disabling the second stage of translation, by changing the HCR.VM bit without changing the VMID

This is consistent with the maintenance required for an ASID and VMID Tagged Virtually Indexed Virtually Tagged (VIVT) instruction cache that also includes a security status bit for each cache entry.

If the IVIPT extensions are implemented, the documented behaviour in [1, section B3.4.1] is unchanged.

Note: if the IVIPT extensions are not implemented, then the ARMv7-A architecture [1, section B3.4.1] requires (amongst other times) that the Instruction Cache is invalidated when enabling or disabling the MMU, by writing to the SCTLR. Therefore, for virtualization, if the IVIPT extensions are not implemented, then switching on the same physical machine from executing in a Non-secure mode other than Hyp mode with a particular VMID with SCTLR.M==0 to executing in a Non-secure mode other than Hyp mode with the same VMID with SCTLR.M==1, or vice versa, all entries in the instruction cache associated with that VMID must be invalidated.

6.2.7 Abort Handling considerations

The Virtualization Extensions create three classes of Data and Prefetch aborts that are taken while executing in Non-secure state

- Those which are taken in to Non-secure Abort mode:
    - Alignment faults, other than those derived as a result of the memory type, while in Non-secure modes other than Hyp mode if HCR.TGE bit is clear
    - Alignment faults derived as a result of the memory type from the first stage translation while in Non-secure modes other than Hyp mode
    - MMU faults other than alignment faults from the first stage of translation while in Non-secure modes other than Hyp mode.

- External aborts (other than during second stage translation table walks) while in Non-secure modes other than Hyp mode if :
    - the SCR.EA bit is clear or the processor is in Debug state, and either
        - the HCR.TGE bit is clear, and the External abort is synchronous, or
        - the HCR.AMO bit is clear, and the External abort is asynchronous
  - Virtual aborts (see section 5.2)
  - Debug exceptions if the HDCR.TDE bit is clear (see section 12.1)
- Those which are taken in Hyp mode:
  - Alignment faults, other than those derived as a result of the memory type, while in Non-secure modes other than Hyp mode if HCR.TGE bit is set
  - Alignment faults derived as a result of the memory type from the second stage translation while in Non-secure modes other than Hyp mode
  - MMU faults other than alignment faults from the second stage of translation while in Non-secure modes other than Hyp mode where the first stage of translation did not cause a MMU fault.
  - MMU faults including alignment faults from the first (and only) stage of translation while in Hyp mode .
  - External aborts if the SCR.EA bit is clear or the processor is in Debug state from
    - Non-secure modes other than Hyp mode and
        - the HCR.AMO bit is set and the External abort is asynchronous, or
        - the HCR.TGE bit is set and the External abort is synchronous
    - Hyp mode
    - Second stage translation table walks (see section 6.2.9)
  - Debug exceptions if the HDCR.TDE bit is set (see section 12.1)
- Those which are taken in Monitor mode:
  - External aborts if the SCR.EA bit is set and the processor is not in Debug state.

In addition, the ARM architecture allows for aborts to be triggered for IMPLEMENTATION DEFINED reasons, associated with Lockdown and Coprocessors. Whether these aborts are directed in Non-secure Abort mode or to Hyp mode is IMPLEMENTATION DEFINED, and an implementation might provide mechanisms to allow the routing of such aborts to be configurable.

In Debug state, the processor does not change mode on taking a Data Abort exception. However, the mode which the abort was *taken in* defines which Fault Status and Fault Address registers are updated by the abort. Aborts in Non-secure Debug state are never taken in Monitor mode.

Prioritization between Aborts:

Multiple Aborts on the same memory access

- If a memory access generates an Alignment fault then that access cannot generate an additional MMU fault from either the first or second stage of translation, cannot generate a Watchpoint debug event, and cannot generate an external abort.
- If a memory access generates an MMU fault from the first stage of translation and an MMU fault from the second stage of translation, the MMU fault from the first stage has priority.
  - If a first stage translation table walk generates a second stage MMU fault (as described in section 6.2.9), the first stage translation table walk does not generate an MMU fault on the first stage of translation.
  - Note that MMU faults of a given stage include synchronous external aborts on translation table walks of that stage of translation

- If a memory access generates an MMU fault from either the first stage or the second stage, then that access cannot generate a Watchpoint Debug event
- If a memory access generates an MMU fault from either the first stage or the second stage or a synchronous Watchpoint Debug event, then that access cannot generate an external abort.
- Other than as stated above, asynchronous external aborts have no architecturally defined priority relationship with any synchronous aborts.

Aborts on different memory transactions generated by the same instruction

- If aborts arise on different memory transactions generated by the same instruction, there is no architecturally defined priority relationship between these aborts.

6.2.7.1 Architectural Fault Status information

For memory system faults taken in Non-secure Abort mode, the architectural fault status information is presented in the Non-secure DFSR for data aborts, and the Non-secure IFSR for prefetch aborts, in the same way as is described in the VMSAv7 in [1] and the Large Physical Address Extensions [2]

For memory system faults taken in Hyp mode (whether generated from Hyp mode or from other Non-secure modes), the architectural fault status information is presented within the Hyp Syndrome Register.

The Hyp Syndrome Register becomes UNKNOWN when executing in Non-secure Kernel or Non-privileged modes except in Debug state.

6.2.7.2 Auxiliary Fault Status Information

The VMSAv7 introduces the concept of Auxiliary Fault Status registers (ADFSR, AIFSR) to hold IMPLEMENTATION DEFINED Fault Status information. Typically, this information would be associated with External aborts (including Parity faults) or with the IMPLEMENTATION DEFINED aborts.

The Virtualization Extensions reserve space in the CP15 register map for an additional set of Auxiliary Fault Status registers, to be used when an abort is taken in Hyp mode if there is a need to have this sort of information. These registers are named the Hyp Auxiliary Data Fault Status Register (HADFSR) and Hyp Auxiliary Instruction Fault Status Register (HAIFSR).

Implementation Note: It is likely in many implementations that all memory aborts that make use of the Auxiliary Fault registers will be taken at the same level of exception. In such an implementation, it is not necessary that two sets of Auxiliary Fault registers are provided, but that instead there is a single set whose designation changes depending on where the fault is configured to be taken.

6.2.7.3 Fault Address Information

The ARM architecture logically provides three sets of fault address registers (IFAR/DFAR – Secure and Non-secure – and HIFAR/ HDFAR) to capture the associated address and status information of these different exceptions. As discussed in section 4.5, the Hyp mode fault address registers are shared with the Secure registers.

For aborts generated and taken in Hyp mode that are caused by the first (and only) stage of translation for Hyp mode, the HIFAR/HDFAR holds the VA that caused the fault in the first stage of translation (regardless of the type of fault – Translation, Access or Permission).

For aborts taken in Hyp mode that are caused by the second stage of translation, the HIFAR/HDFAR holds the VA that caused the fault in the second stage of translation (regardless of the type of fault – Translation, Access or Permission).

For aborts taken in Hyp mode that are caused by Translation or Access faults in the second stage of translation, or caused by faults in the second stage of translation during a translation table walk of the first stage translation (which could be either a translation, access or a permission fault), the faulting IPA is held in the HPFAR register. For other aborts, this register is UNKNOWN.

For aborts taken in Hyp mode that are caused by Synchronous External aborts the HIFAR/HDFAR holds the VA that caused the fault.

The HIFAR is UNKNOWN on a Data abort taken in Hyp mode. The HDFAR is UNKNOWN on a Prefetch Abort taken in Hyp mode. The HIFAR and HDFAR become on UNKNOWN when executing in Non-secure Kernel or Non-privileged modes.

For Data aborts taken in Hyp mode that are caused by watchpoints, the HDFAR holds the VA that generated the watchpoint. This is consistent with the handling of watchpoints using DFAR described in [1].

Prefetch aborts taken in Hyp mode that are caused by Debug exceptions do not make use of the HIFAR but leaves it UNKNOWN.

For Watchpoint debug exceptions when the HDCR.TDE bit is set, the DBGWFAR is based on the VA of the instruction that accessed the watchpointed location for asynchronous watchpoints. The DBGWFAR is UNKNOWN for asynchronous watchpoints.

6.2.7.4 TLB Caching restrictions for Second Stage Translation Table Entries

In keeping with the behaviour of the ARMv7 VMSA, any translation table entries which themselves cause a Translation Fault or Access Fault are not allowed to be held in a TLB, so that if these entries are changed, there is no need to perform TLB Maintenance. Second stage translation table entries that cause permission faults are allowed to be held in a TLB, and so do require TLB maintenance.

6.2.8 CP15 Register Consequences of Supporting Two Stages of Translation and of Virtualization in general

6.2.8.1 System Control

6.2.8.1.1 Control for second stage of translation control

A control bit is required which is logically equivalent to the SCTLR.M bit, but applying to the second stage of translation. When this bit is low (the reset state) the translation from IPA to PA is a flat mapping, with no effect on the memory or permission attributes. This bit is held in HCR, the HCR.VM bit, being a new bit introduced for the Virtualization Extensions.

If the HCR.VM to be used when executing in a Non-secure mode other than Hyp mode with a particular VMID is changed then the entries in the TLB associated with that VMID must be invalidated for effect of the change of the HCR.VM to be guaranteed to be visible when in a Non-secure mode other than Hyp mode with that VMID.

6.2.8.2 Second Stage Translation Base and Control Registers

The equivalent for the TTBR0 and TTBCR registers for the second stage of translation are called the VTTBR and VTCR respectively and are added as part of the Virtualization Extensions. As the second stage translation is capable of mapping to a > 32 bit physical address, the VTTBR register is a 64-bit register accessed using MCRR/MRRC instructions. The VTTBR also holds the VMID as an 8-bit field.

These registers are R/W, and can be accessed in Secure state and in Hyp mode only. Attempting to access these registers in Non-secure state in modes other than Hyp mode is UNDEFINED.

6.2.8.3 Data/Unified Cache maintenance by MVA

In the ARMv7 Architecture, the full virtual address passed on a cache maintenance instruction is generally taken to be a combination of the current ASID, current Security State and the MVA that is passed with the instruction. In the data operations, this full address is translated by the MMU to give the physical address on which the instruction must operate. This principle is extended within the Virtualization Extensions, where the full virtual address now incorporates whether the instruction was performed in a Hyp mode or one of the Non-secure Kernel modes, and, when executed from one of the Non-secure Kernel modes, the VMID.

Data Invalidate by MVA executed from the Non-secure state other than in Hyp mode must not be able to cause a change to data in locations that do not have write access permission in the second stage permissions.

In this case, it is IMPLEMENTATION DEFINED whether:

- DCIMVAC is transformed into a DCCIMVAC in cases where a permission violation occurs OR
- A second stage permission fault is generated for DCIMVAC

Implementation Note: It is always functionally acceptable to implement DCIMVAC as DCCIMVAC while in the Non-secure state and the exact occasions when this is performed are therefore implementation specific. Implementation approaches could involve converting a DCIMVAC to DCCIMVAC for operations performed in the Non-secure state whenever the second stage translations are being used, or only when a 2nd stage permission failure is detected.

Data/Unified cache maintenance operations by MVA executed within the Hyp mode or from Secure state are unchanged by virtualization.

6.2.8.4 Data/Unified Cache maintenance by Set/Way

The architecture does not require a cleaning of the cache when switching between different virtual machines. For this reason, the Data/Unified Cache Invalidation by Set/Way must not present an opportunity for one virtual machine to corrupt the state of a second virtual machine. The Virtualization Extensions provide a control in the HCR (the HCR.SWIO bit) which, when set, forces Invalidate by Set/Way in Non-secure Kernel modes to be treated as Clean and Invalidate by Set/Way.

6.2.8.5 Instruction Cache maintenance by MVA

In the ARMv7 Architecture, the full address passed on a cache maintenance instruction is derived from combination of the current ASID, current Security State and the MVA that is passed with the instruction. This principle is extended within the Virtualization Extensions, where the full virtual address now incorporates whether the instruction was performed in a Hyp mode or one of the Non-secure Kernel modes, and, when executed from one of the Non-secure Kernel modes, the VMID.

Note: The function of the instruction cache maintenance operations inherently reveal whether or not it is virtually indexed, or even virtually tagged, as the cache maintenance operations by address are required to only have an effect within the context that they are issued (see [1, section B3.4.2]). This principle is unchanged by the Virtualization Extensions.

Note: The incorporation of the VMID into the full virtual address when executing from Non-secure Kernel modes applies even at times when the HCR.VM bit is not set. For this reason, the VMID is defined to be reset to 0, which allows the incorporation of the VMID to be invisible in situations where the VMID is not changed from reset.

6.2.8.6 Instruction Cache Invalidate All

The instruction cache invalidate all operation are permitted to apply to all unlocked entries within the instruction cache, but are required to apply to all entries that are relevant to the software component that executed them.

- If these instructions are executed from a Non-secure Kernel mode, these instructions are only required to apply to Non-secure entries associated with the current Virtual Machine (with the same VMID).
- If these instructions are executed from Non-secure Hyp mode, these instructions are only required to apply to all Non-secure entries.
- If these instructions are executed from a Secure Privileged mode, these instructions are required to apply to all entries.

ICIALLU can be forced by the hypervisor to be broadcast within the Inner Shareable domain (so upgraded to ICIALLUIS) when executed from the Non-Secure Kernel modes. This is controlled by the HCR.FB bit.

6.2.8.7 Lockdown of Caches

While the mechanisms for cache lockdown are no longer architected, the limits of the behaviour of lockdown have been described within the architecture. In general it is expected that, as cache lockdown is about the control of actual physical cache lines, the control of lockdown must ultimately be controlled by the hypervisor or from Secure code. A number of possible strategies can be envisaged for lockdown:

1. The use of lockdown is reserved for functionality accessed from the Secure state.
2. The use of lockdown is reserved for the hypervisor
3. One particular virtual machine is granted the right by the hypervisor to use the lockdown facilities
4. The hypervisor arbitrates between different lockdown requests by different virtual machines.

The interaction of cache maintenance operations with lockdown is covered by the architecture – the principles outlined in the ARM ARM Section B2.2.5 are unchanged by the addition of virtualization.

Exceptions triggered by lockdown related problems when executing in the Kernel modes in Non-secure state can be configured to be taken in either

- Non-secure abort mode using the Non-secure Data abort vectors or
- Hyp mode, using the equivalent of the Data abort vector

This configuration is controlled by the HCR.TLD bit.

6.2.8.8 TLB maintenance instructions

6.2.8.8.1 Instructions defined before the Virtualization Extensions

These instructions can be executed in Non-secure Kernel modes, Hyp mode and in Secure Privileged modes. This list is unchanged from the set described in [1]:

Invalidate entire Data TLB	DTLBIALL
Invalidate Data TLB entry by MVA	DTLBIMVA
Invalidate Data TLB entry by ASID match	DTLBIASID
Invalidate entire Instruction TLB	ITLBIALL
Invalidate Instruction TLB entry by MVA	ITLBIMVA
Invalidate Instruction TLB entry by ASID match	ITLBIASID
Invalidate entire Unified TLB	TLBIALL
Invalidate Unified TLB entry by MVA	TLBIMVA
Invalidate Unified TLB entry by ASID match	TLBIASID
Invalidate Unified TLB entry by MVA All ASID	TLBIMVAA
Invalidate entire Unified TLB inner shareable	TLBIALLIS
Invalidate Unified TLB entry by MVA inner shareable	TLBIMVAIS
Invalidate Unified TLB entry by ASID match inner shareable	TLBIASIDIS
Invalidate Unified TLB entry by MVA All ASID inner shareable	TLBIMVAAIS

Note – the Data and Instruction operations were deprecated in ARMv7 MP Extensions and can be implemented as aliases of the Unified TLB operations.

These instructions are permitted to apply to all unlocked entries within the TLB, but are required to apply to the following entries:

- If these instructions are executed from a Non-Secure Kernel mode, these instructions are only required to apply to Non-secure entries associated with Non-Secure Kernel and Non-Privileged modes with the current VMID.
- If these are executed from Non-secure Hyp mode, these instructions are only required to apply to all Non-secure entries associated with Non-Secure Kernel and Non-Privileged modes with the current VMID
- If these instructions are executed from a Secure Privileged mode, these instructions are only required to apply to entries associated with execution in Secure state

Note: The use of the VMID applies even in situations when the HCR.VM bit is clear, including situations when there is no use of Virtualization. Provided that the VMID value is unchanged from its reset value, this does not introduce any backwards compatibility issues.

Note: The TLB operations that use the MVA still function in situations where the SCTLR.M bit is 0.

The TLB Maintenance instructions which are not broadcast within the Inner Shareable domain can be forced to be broadcast within the Inner Shareable domain when executed from the Non-Secure Kernel modes by the hypervisor. This is controlled by the HCR.FB bit.

6.2.8.8.2 New Instructions defined by the Virtualization Extensions

The following operations are introduced as part of the Virtualization Extensions, and are available in Hyp mode and in Secure state only and are UNDEFINED otherwise:

Invalidate Unified Hyp TLB entry by MVA	TLBIMVAH
Invalidate Unified Hyp TLB entry by MVA inner shareable	TLBIMVAHIS
Invalidate entire Non-secure Non-Hyp Unified TLB	TLBIALLNSNH
Invalidate entire Non-secure Non-Hyp Unified TLB inner shareable	TLBIALLNSNHIS
Invalidate entire Hyp Unified TLB	TLBIALLH
Invalidate entire Hyp Unified TLB inner shareable	TLBIALLHIS

The Hyp instructions are only required to apply to Non-secure entries associated with execution in Hyp mode. The argument passed for these instructions has the ASID field as SBZ.

The Non-Hyp instructions are only required to apply to Non-secure entries associated with execution in Kernel and Non-Privileged modes for all VMIDs

The use of these instructions from Secure Privileged modes other than Monitor mode is UNPREDICTABLE.

Encodings for these operations are as follows:

TLBIMVAH:	MCR p15, 4, Rt, C8, C7, 1
TLBIMVAHIS:	MCR p15, 4, Rt, C8, C3, 1
TLBIALLH:	MCR p15, 4, Rt, C8, C7, 0
TLBIALLHIS:	MCR p15, 4, Rt, C8, C3, 0
TLBIALLNSNH:	MCR p15, 4, Rt, C8, C7, 4
TLBIALLNSNHIS:	MCR p15, 4, Rt, C8, C3, 4

6.2.8.9 Lockdown of TLB

The treatment in the architecture of TLB lockdown is essentially similar to that of cache lockdown, as described in section 6.2.8.7

The interaction of TLB maintenance operations with lockdown is covered by the architecture – the principles outlined in the ARM ARM Section B3.10.4 are unchanged by the addition of virtualization.

Exceptions triggered by lockdown related problems when executing in the Kernel modes in Non-secure state can be configured to be taken in either

- Non-secure abort mode using the Non-secure Data abort vectors, or
- Hyp mode, using the vector offset 0x14

This configuration is controlled by the HCR.TIDCP bit.

6.2.8.10 VA to PA operations

The currently architected VA to PA instructions on the current side for operations which apply to the memory translations used when executing in Non-secure modes other than Hyp mode are redefined in the Virtualization Extensions to perform VA to IPA translations, presenting the IPA information in the Non-secure PAR of the security state from which they were executed.

From the Secure state, the existing VA to PA operations on the same state (V2PCW**) perform VA to PA translations, as the transaction issued from the secure state have only one stage of translation.

The currently architected VA to PA instructions on the “other side”, that can be executed with in the Secure state to determine the translations used in the Non-secure state are defined to apply to translations used when executing in Non-secure modes other than Hyp mode. These instructions perform VA to PA translations, incorporating both stages of translation.

A new set of instructions is provided that provide the VA to PA translations that are used when executing in Hyp mode. These instructions are accessible in Hyp mode and Secure Privileged modes only. The use of the new instructions from Secure Privileged modes other than Monitor mode is deprecated.

Note: The introduction of these instructions into Secure Privileged modes other than Monitor mode is for consistency of access permission with respect to the existing instructions. The deprecation of their use in Secure Privileged modes other than Monitor mode is similarly for consistency with those other instructions.

Where the address reported in the PAR can be larger than 32 bits, the PAR is extended to be a 64-bit register. This is discussed in [2].

Note: the mnemonics and naming of these instructions becomes a little illogical and potentially confusing as a result of the introduction of different levels of address space. The existing mnemonic names need to remain for backwards compatibility, but a new set of mnemonics, based on the stage of translation are preferable. Such a naming strategy would be as shown in the following table

Current Name	New Name	New Function	Executed from
V2PCW**	ATS1C**	Address Translation Stage 1 Current state: In Secure state: translations used by memory accesses in Privileged and non-Privileged modes In Non-Secure state: translations used by memory accesses in Kernel and non-Privileged modes	Secure Privileged Modes, Hyp Mode and Non-secure Kernel Modes
V2POW**	ATS12NSO**	Address Translation Stage 1 and 2 translations used by memory accesses in Kernel and non-Privileged modes in the Non-Secure state	Secure Privileged Modes and Hyp Mode only (see access restriction below)
N/A	ATS1H[R,W]	Address Translation Stage 1 translations used in Hyp mode	Secure Monitor Mode and Hyp Mode only (see access restriction below)

Access Restriction: The use of ATS12NSO** instructions from Secure Privileged modes other than Monitor mode is deprecated.

The use of ATS1H[R,W] instructions from Secure Privileged modes other than Monitor mode is UNPREDICTABLE.

The encodings for the new instructions are as follows;

ATS1HR MCR p15, 4, Rt, C7, C8, 0

ATS1HW MCR p15, 4, Rt, C7, C8, 1

6.2.8.10.1 Format of the PAR

The format of the PAR is changed in the LPAE specification to accommodate a larger Physical address and additional attribute encodings when the new translation table format is used and the mode that is used.

For the ATS1C** instructions executed in Non-secure Kernel or Secure Privileged modes, the format used depends on the TTBCR.EAE bit for the Security state being translated.

For the ATS1C** instructions executed in Hyp mode, the new format is always used.

For the ATS1H* instructions the new PAR format is always used. For the ATS12NSO** operations, the choice of PAR format is determined by the Non-secure TTBCR.EAE bit and the HCR.VM bit:

- When Non-secure TTBCR.EAE == 0 and HCR.VM ==0, the ARMv7 format of the PAR, a 32-bit register, is used
- Otherwise the new, 64-bit, register is used.

The Physical Address is 0 extended if it comes from a 32-bit translation and is reported in the 64-bit format PAR.

For the ATS12NSO** operations, the memory and shareability attributes reported are the combined attributes for stage 1 and 2.

6.2.8.10.2 Aborts during VA to PA translation operations

In the ARMv7 specification, if a VA to PA operation comes across a situation that would abort, other than an external abort, the abort is not taken, but the presence of the abort is captured within the PAR, together with the Fault Status information.

The same principle applies in the Virtualization Extensions with one exception. For the ATS1C** instructions executed in a Non-secure Kernel mode, if there is a second stage fault (translation, access or permission fault or a synchronous external abort on a second stage translation table walk) during a first stage translation table walk, the fault is taken as an abort into the Hyp mode, and the PAR is UNKNOWN.

In this case:

- The HSR syndrome register indicates that the fault occurred on a translation table walk and that the operation was a cache maintenance operation.
- The HPFAR holds the IPA that faulted.
- The HDFAR holds the Virtual Address that was requested to be translated.

6.2.9 Faults During First Stage Translation Table Walks

Translation table walks performed as part of the First Stage translation use addresses that are subject to translation by the second stage of translation. There are two possible consequences of this:

1. The first stage translation table walks can give rise to a translation, access or permission fault in the second stage of translation. They can also give rise to synchronous external aborts on a second stage translation table walk. These are all treated second stage memory aborts; a bit in the HSR is used to distinguish this source of fault, and the part of the syndrome field containing details of the instructions is invalid.
2. The first stage translation table walks could be to an area of memory with the Device or Strongly-Ordered memory attribute described within the second stage of translation. In most systems, this is likely to be indicative of some sort of error within the Guest OS, where the first stage translation is corrupted. A configuration setting is provided in the HCR, the HCR.PTW bit to allow this to be faulted as a second stage permission fault.

The effect of the HCR.PTW might be held in entries in a TLB associated with a particular VMID. Therefore if the HCR.PTW bit changed without a change in VMID, it is necessary to invalidate all entries associated with that VMID before executing in a Non -secure mode other than Hyp mode, or else the behaviour is UNPREDICTABLE.

6.2.10 Stage 1 Default Memory Type

When the Stage 1 MMU is disabled, the ARMv7 VMSA requires that all data memory accesses are to Strongly-Ordered memory. This is because, when the Stage 1 MMU is disabled, in a system without a second stage of translation, the VMSA would otherwise not be able to distinguish which memory regions can be cached, or speculated to, or re-ordered.

When the second stage of translation is present, it becomes feasible for the memory types to be distinguished using the second stage of translation. In such a situation, a simple Guest Operating System could avoid needing to have any first stage translation capability and so have reduced context switch costs. However, given the approach to handling memory attributes described in section 6.2.3, the default memory type for this sort usage is best set to the most “permissive” – that is: Non-Shareable, Inner and Outer WriteBack, Write Allocate. This applies to both instruction and data accesses.

This is only feasible when the Hypervisor is working in conjunction with the Guest Operating System to use such functionality. For this reason, the control for the setting the default memory type is held within the HCR.

When the HCR.DC bit is set, the memory type determined when the stage 1 MMU is disabled, when executing in Non-Secure modes other than Hyp mode, is set to be Non-Shareable, Inner WriteBack WriteAllocate, Outer WriteBack WriteAllocate. When the HCR.DC bit is not set, the memory type when the stage 1 MMU is disabled when executing in Non-Secure modes other than Hyp mode is unchanged from that defined in the [1, Section B3.2.3]. The effect of the HCR.DC bit applies to memory type assigned by the stage 1 MMU, and this memory type is then affected by the second stage as described in section 6.2.3.

The effect of the HCR.DC bit is reflected in the results returned from an ATS1C** or ATS12NSO** instruction when it accesses the translation used in the Non-secure state and the Non-secure SCTLR.M bit is clear.

Executing in a Non-secure mode other than Hyp mode with both SCTLR.M set and HCR.DC bit set is UNPREDICTABLE.

Executing in a Non-secure mode other than Hyp mode with HCR.VM clear and HCR.DC bit set is UNPREDICTABLE.

The effect of the HCR.DC might be held in entries in a TLB associated with a particular VMID. Therefore if the HCR.DC bit changed without a change in VMID, it is necessary to invalidate all entries associated with that VMID before executing in a Non -secure mode other than Hyp mode, or else the behaviour is UNPREDICTABLE.

6.2.11 Alignment Handling

Alignment faults that are generated as a result of:

- the SCTLR.A bit being set
- An unaligned address for an instruction that does not support unaligned addresses

when executing in Non-secure Kernel modes and in Non-secure Non-privileged modes are taken in Non-secure Abort mode.

In the Virtualization Extensions, if an unaligned access is performed to an area of memory with the Strongly-Ordered or Device memory attribute, an Alignment Fault is generated. This is new requirement for the Virtualization Extensions, as previously the behaviour was UNPREDICTABLE; however most implementations had implemented this as an abort.

When executing in Non-secure Kernel modes and in Non-secure Non-privileged modes, then:

- if the stage 1 translations mark the region of memory as Strongly-Ordered or Device, the abort is taken in Non-Secure Abort mode.
- if the stage 2 translations mark the region of memory as Strongly-Ordered or Device, the abort is taken in Hyp mode.

In the event that an alignment fault is caused by multiple factors (for example, the SCTLR.A bit is set, and the access is marked as Strongly-Ordered), then the abort is taken is taken in Non-secure Abort mode if any of the factors require that the abort is taken in Non-secure Abort mode.

An alignment fault associated with a particular stage is prioritised between

- AccessFlag fault and Domain fault for that stage if the translation system can generate Domain faults
- AccessFlag fault and Permission fault for that stage if the translation system cannot generate Domain faults

In all cases, Alignment faults in Hyp mode are taken in Hyp mode. Alignment faults in Secure modes are taken in Secure Abort mode.

6.3 Exposing the MMU to Other Masters

With the explicit recognition of two stages of translation within the architecture, some aspects of device handling are made simpler. The architecture now provides a defined structure that devices, driven in intermediate physical space, can use to obtain physical addresses. Concretely, this requires the existence of a “system MMU”. With the use of such things, the option to introduce virtually addressed devices (eg Virtual DMA) becomes more feasible. Much of this is beyond the normal scope of the ARM architecture, but derives strongly from it. ARM intends to drive a standard for such a component with the details expected to be published in 1H-2011.

7 EMULATION SUPPORT

7.1 Overview

Virtualization tends to require the hypervisor to emulate some instructions. The reasons for this are:

- The need for trapped loads and stores associated with accesses to virtual devices to be interpreted by the hypervisor. These loads and stores are often emulated by the hypervisor.
- The need for trapped accesses to some system control registers to be emulated by the hypervisor. In general the number of accesses that need to be trapped should be kept to a minimum, but it is not necessarily possible to avoid some need to trap to the hypervisor.

When an instruction has to be emulated, there a number of steps that must be performed for each instruction being emulated:

- Reading of the instruction - this is complicated by the fact that the instruction is held in the address space of the Guest OS, and so getting the address of the instruction involves translating the virtual address held in the Exception Link Register into an address that can be used by the hypervisor.
- Decoding the instruction, allowing for the fact that the instruction may be an ARM, Thumb or ThumbEE instruction; the SPSR.T and J bits are used to determine this.
- Performing the emulation of the instruction

While each of these steps is relatively straightforward, the combination of the steps can result in a substantial overhead for emulating each trapped instruction. To reduce this overhead, the Virtualization Extensions introduce syndrome information in a register called the Hyp Syndrome Register (HSR) for the common cases of emulation to reduce the penalties involved.

7.2 Load/Store Emulation support

Presenting virtual device interfaces requires the hypervisor to trap loads and stores to the virtual device. It is anticipated that this will be performed using a Stage2 abort, and this probably best done using a permission failure to allow it to be cached in the TLB.

The load or store could, in principle, be a multiple, or have writeback, so there are many complex cases to handle for this emulation.

The simple cases (single loads/stores – byte/half-word/word – without writeback) are by far the most common in device drivers. The emulation task can be accelerated by presenting more syndrome information relating to the instruction that caused the fault within the HSR. This extended syndrome contains the following information:

- Valid
- Access Size of transaction
- Sign extension of the transaction
- Source/Target register
- Size of aborted instruction

If the extended syndrome is not valid, the instruction must be read and decoded to allow it to be emulated.

The extended syndrome information is valid for the following instructions when

- The instruction does not do register writeback, or
- When the destination is not R15

  LDR, LDRT, LDRSH, LDRSHT, LDRH, LDRHT, LDRSB, LDRSBT, LDRB, LDRBT, STR, STRT, STRH, STRHT, STRB, STRBT

The extended syndrome information is not valid for any other instructions.

Note: in the ARM instruction set, the LDR*T and STR*T instructions all perform register writeback and so the extended syndrome is not valid in these cases.

7.3 Other Emulation support

The same difficulty in reading the instruction to be emulated exists for any other instructions which need to be trapped to the hypervisor and emulated. At present, these are mostly associated with accesses to system registers using CP15 MCR/MRC instructions, and importantly are not data dependent.

The HSR holds syndrome information of the exception which caused the trap to the hypervisor. This avoids the hypervisor having to interpret the virtual address of the instruction and read the instruction as part of its emulation handler.

The HSR is used for:

- All Exceptions other than Interrupts that cause an entry into Hyp mode
- HVC or SVC when executed in any Non-Secure mode (including Hyp mode) that cause an entry into Hyp mode
- All Aborts which are generated in Hyp mode and are taken in Hyp mode
- All UNDEFINED exceptions which are generated in Hyp mode and are taken in Hyp mode. In some cases, these exceptions are not categorised.

Note: A syndrome, rather than the aborted instruction, is presented to allow for implementations where the details of the instruction are discarded early in the pipeline.

8 CONFIGURABLE TRAPS TO THE HYPERVISOR

8.1 Overview

It is necessary in a full virtualization solution to able to trap attempts to access functionality of the processor which the hypervisor needs to intercept. Typically this is where the hypervisor will provide some different functionality than the native hardware in response to the accessing instructions. Mostly this applies to accesses to system control state, such as CP15 registers.

The mechanism for this trapping is the taking of an exception into Hyp mode (as opposed to entering into the Non-secure UNDEFINED mode), going to the vector offset used for entries into Hyp mode (0x14), so the vector address is therefore Hyp Vector Base Address + 0x14 in all cases. The HSR register holds syndrome information for such faults. This is referred to as taking an Hyp Entry Trap.

All such traps are treated as synchronous exceptions that prevent the instruction that is being trapped from executing. The state presented to the exception handler is therefore consistent with the state of the system immediately before the instruction that was trapped.

Most of these traps apply only for accesses from the Kernel modes in Non-secure state only; a small number apply also to User mode in Non-secure state:

- Most registers involved are described in [1] as being not accessible from User mode in Non-secure state, but instead generate an UNDEFINED exception when executed in Non-secure User mode, which is taken into the UNDEFINED mode. This behaviour in unchanged by the Virtualization Extension.
- If a register is described in [1] as being accessible from User mode in Non-secure state, and it has a trap added to Hyp mode, then the trap to Hyp mode applies for Kernel and Non-Privileged accesses in Non-secure state.

Unless otherwise stated, Traps to Hyp mode do not apply:

- When executing in Hyp mode.
- When executing in Secure states, regardless of the value of the SCR.NS bit.
- When in Debug State

Note: The HCPTR, described in section 8.14, has an additional role when in Hyp mode.

As the set of situations which need to be trapped varies between different use models of Virtualization, a number of control bits are used within the Hyp Configuration Register and Hyp System Trap Register to control at a suitable level of granularity which situations need to be trapped.

8.1.1 Hyp Traps on Instructions that Fail Their Condition Codes

For instructions with traps set that fail their condition code check, unless otherwise stated, it is IMPLEMENTATION DEFINED whether these instructions take the trap or behave as a NOP. This is consistent with the treatment of conditional UNDEFINED instructions described in the [1 section B1.6.11].

8.1.2 Hyp Traps on Instructions that are UNPREDICTABLE

For instructions which are UNPREDICTABLE, but where the instruction is in a class that have Hyp traps, the behaviour of these instructions is UNPREDICTABLE.

Note: it is permissible, but not required, that such instructions cause Hyp Traps.

Note: the definition of UNPREDICTABLE in preventing execution at a particular level from being able to perform any action that cannot be performed at the level of privilege using instructions that are not UNPREDICTABLE means that setting a Hyp Trap on an instruction changes the set of instructions that could be executed at user or kernel levels of privilege and so indirectly influences the set of behaviours that are permissible from UNPREDICTABLE instructions.

Note: If no instructions are configured to take Hyp Traps, then an UNPREDICTABLE instruction at Kernel or User mode cannot generate Hyp Traps.

8.1.3 Hyp Traps on Instructions that are UNDEFINED

In general, and except where otherwise stated in specific cases in this section, if an instruction is executed in a Non-secure mode other than Hyp mode that would cause an Undefined Exception and also has a Hyp trap associated with it, the resulting behaviour is to take the Undefined Exception.

8.2 ID Mechanisms

The ID mechanisms in register 0 that are accessible in Non-secure state present a virtualization hole as it is possible for the Guest OS to determine information about the underlying hardware which the hypervisor might want to conceal.

In many uses of virtualization there is no need for the hypervisor to be able to disguise the identity of the physical processor.

As a result, the Virtualization Extensions split the ID mechanisms into a number of groups, and to provide separate control bits within the HCR to determine whether or not accesses to the registers are trapped to the hypervisor.

8.2.1 Writing the ID registers used by the Guest OS

For some commonly accessed ID registers, a different virtualization approach from trapping to the hypervisor is adopted. In this approach, the register being read when in the Kernel modes in Non-secure state is a register that has been written by the hypervisor. This involves the creation of a read/write register accessible from Hyp mode and Monitor mode when SCR.NS==1 which is read in place of the relevant ID register when in the Kernel modes in Non-secure state. The reset value of the read/write register is made the same as the ID register that it substitutes.

This approach is relatively area expensive, so it is used for a small number of registers which are expected to be accessed relatively frequently:

MIDR

MPIDR

For registers treated this way, all reads from Hyp mode and from the Secure state (regardless of the value of SCR.NS) to the read-only register address read the physical register.

8.2.2 Grouping of ID Registers

8.2.2.1 MPIDR

For multi-processor implementations, the MPIDR provides information as to the CPU position within a cluster, and it is likely that this needs to be virtualized by the hypervisor. The virtualization of this register is performed using the technique described in section 8.2.1.

The value returned by a Non-secure Kernel mode read of the MPIDR is held in the VMPIDR register, which is described in section 13.16.

8.2.2.2 MIDR

The MIDR provides information about the implementor and processor name and revision information. It is likely that this needs to be virtualized by the hypervisor. The virtualization of this register is performed using the technique described in section 8.2.1.

The value returned by a Non-secure Kernel mode read of the MIDR is held in the VPIDR register, which is described in section 13.15.

8.2.2.3 Primary Device Identification Registers (ID Group 0)

This set of registers comprises the coarse grained identification mechanisms which do not require the technique described in section 8.2.1.

FPSID – Floating-point System ID Register

JIDR – Jazelle ID Register

Any attempt that is not UNDEFINED in [1], to read the registers listed above from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TID0 bit is set.

If the FPSID is also trapped by the HCPTR, then the HCPTR trap has priority.

8.2.2.4 Implementation Identification Registers (ID Group 1)

This set of registers comprises the coarse grained identification mechanisms that from ARMv7 mostly contain implementation specific information

TCMTR – TCM Type Register

TLBTR – TLB Type Register

AIDR – Auxilliary ID Register

Any attempt that is not UNDEFINED in [1], to read the registers listed above from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TID1 bit is set.

8.2.2.5 Cache Identification Registers (ID Group 2)

This set of registers comprises the cache information registers

CTR - Cache Type Register

CCSIDR – Cache Size ID Registers

CLIDR – Cache Level ID Register

CSSELR – Cache Size Selection Register

Any attempt that is not UNDEFINED in [1], to read the registers listed above from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TID2 bit is set.

8.2.2.6 Detailed Feature Identification (ID Group 3)

This set of registers comprises the detailed CPUID register information:

ID_PFR0 – Processor Feature Register 0

ID_PFR1 – Processor Feature Register 1

ID_DFR0 – Debug Feature Register 0

ID_AFR0 – Auxiliary Feature Register 0

ID_MMFR0 – Memory Model Feature Register 0

ID_MMFR1 – Memory Model Feature Register 1

ID_MMFR2 – Memory Model Feature Register 2

ID_MMFR3 – Memory Model Feature Register 3

ID_ISAR0 – Instruction Set Attribute Register 0

ID_ISAR1 – Instruction Set Attribute Register 1

ID_ISAR2 – Instruction Set Attribute Register 2

ID_ISAR3 – Instruction Set Attribute Register 3

ID_ISAR4 – Instruction Set Attribute Register 4

ID_ISAR5 – Instruction Set Attribute Register 5

MVFR0 - Media and VFP Feature Register 0

MVFR1 – Media and VFP Feature Register 1

Any attempt that is not UNDEFINED in [1], to read the registers listed above from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TID3 bit is set.

If the MVFR0 or MVFR1 is also trapped by the HCPTR, then the HCPTR trap has priority.

8.3 Implementation Defined CP15 Features

The implementation defined features in CP15 covering features of the ARM architecture such as lockdown, DMA and tightly-coupled memory, are constrained to the following registers:

All CP15, CRn==9, Opcode1 = {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}

All CP15, CRn==10, Opcode1 =={0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}

All CP15, CRn==11, Opcode1=={0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}

Any attempt to access the registers listed above from a Non-secure Kernel Mode causes a Hyp Entry Trap when the HCR.TIDCP bit is set.

This bit also determines that any lockdown faults in the memory system are taken as Data Aborts in Hyp mode.

It is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure User Mode causes a Hyp Entry Trap when the HCR.TIDCP bit is set; alternatively it will result in an Undefined exception to be taken in Non-Secure UNDEFINED mode.

Notes:

- Implementations might provide additional controls in the HACR implementation defined register (see section 13.23) to provide finer grained control of control of trapping of IMPLEMENTATION DEFINED features.
- The trapping of Non-secure User Mode access to this functionality to Hyp mode is expected to be unusual, and should be used where the hypervisor is virtualizing user mode functionality. In most cases, it is expected that Non-secure User Mode access to this functionality will result in an UNDEFINED exception taken in Non-secure Undefined mode, as would be the case in the equivalent system without the virtualization extensions.
- The trapping of all attempted accesses to these registers from a Non-secure Kernel Mode overrides the general behaviour described in section 8.1.3

8.4 Use of Data Cache Maintenance by Set/Way Instructions

A trap is provided to trap the use of Data Cache Maintenance by Set/Way instructions to the hypervisor:

DCISW, DCCSW, DCCISW

Any attempt to execute these instructions from a Non-secure Kernel Mode causes a Hyp Entry Trap when the HCR.TSW bit is set.

Note: The Set/Way data cache maintenance instructions are problematic for virtualization of uniprocessor systems within an MP system, where a virtual machine can be moved between different physical processors. This is because the Set/Way operations can be interrupted part way through their operation and the effect must be reproduced on both processors.

8.5 Use of Data Cache Maintenance to Point of Coherency Instructions

A trap is provided to trap the use of the data cache maintenance instructions by MVA to the point of coherency to the hypervisor. The instructions involved are:

DCIMVAC, DCCIMVAC, DCCMVAC

Any attempt to execute these instructions from a Non-secure Kernel Mode causes a Hyp Entry Trap when the HCR.TPC bit is set.

8.6 Use of Cache Maintenance to Point of Unification Instructions

A trap is provided to trap the use of the cache maintenance instructions to the point of unification to the hypervisor. The instructions involved are:

ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAU

Any attempt to execute these instructions from a Non-secure Kernel Mode causes a Hyp Entry Trap when the HCR.TPU bit is set.

8.7 Use of TLB Maintenance Instructions

A trap is provided to trap the use of the TLB maintenance instructions to the hypervisor. The instructions involved are:

TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBMVAAIS, *TLBIALL, *TLBIMVA, *TLBIASID, TLBMVAA,

Any attempt to execute these instructions from a Non-secure Kernel Mode causes a Hyp Entry Trap when the HCR.TTLB bit is set.

8.8 Accesses to Auxiliary Control Register Functionality

The ACTLR is an IMPLEMENTATION DEFINED register which is permitted to implement global control bits. Any attempt to access this by the Guest OS indicates that there is a potential virtualization problem with that Guest OS. Trapping these operations to the hypervisor allows the hypervisor to react, either by performing some emulation of the function, or by signalling a virtualization error.

Any attempt that is not UNDEFINED in [1], to access the ACTLR from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TAC bit is set.

8.9 Accesses to Performance Monitor Functionality

The Performance Monitor functionality may be allocated to a particular Guest, or may be virtualized by the hypervisor. A first trap bit, HDCR.TPM, is provided to allow trapping of attempts to access the Performance Monitors; this can be used as part of a lazy context switching for allocating performance monitoring to a single Guest, or as part of a virtualization approach.

The Performance Monitor functionality is allocated in the following register ranges:

All CP15 CRn==9, opcode1={0-7}, CRm == {c12-c15}, opcode2 == {0-7}

Any attempt that is not UNDEFINED in [1], [4] or [5], to access these from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HDCR.TPM bit is set.

A second trap bit, HDCR.TPMCR, is provided to allow trapping of attempts to access the Performance Monitor Control Register (PMCR); this can be used to emulate the PMU identification bits contained in that register.

The PMCR is accessed by the following instructions:

```
MRC    p15,0,Rt,c9,c12,0        ; Read PMCR
MCR    p15,0,Rt,c9,c12,0        ; Write PMCR
```

Any attempt that is not UNDEFINED in [1], [4] or [5], to access the PMCR from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HDCR.TPMCR bit is set.

8.10 Use of SMC in Non-secure Kernel Modes

In a system using a hypervisor, it is desirable that the hypervisor can determine which, if any, virtual machines have direct access to the Secure state, and so can execute an SMC instruction. To this end, a new bit, the HCR.TSC is added. When the HCR.TSC bit is set, and the core is executing in Non-secure Kernel mode, any attempt to execute an SMC instruction causes a Hyp Entry Trap.

If this trap is enabled at the same time as the SCR.SCD bit, then the Hyp Entry Trap is taken if the SMC instruction is executed in Non-secure Kernel modes.

Conditional SMC instructions are only trapped by this mechanism if the instruction that causes the exception passes its condition codes.

8.11 WFI and WFE handling

WFI is used as a mechanism by a Guest OS to signal that it should suspend operation pending an interrupt. In a virtualized system it is often desirable for the hypervisor to know that the Guest OS is in this state, so that another Guest OS can be run. In general it is expected that WFI will involve suspension of a Guest OS for a sufficient period of time that it is worth scheduling a different Guest OS. For this reason it is useful to be able to trap WFI to the hypervisor.

When the HCR.TWI bit is set and the core is executing in Non-secure state other than Hyp mode then the execution of a WFI causes a Hyp Entry Trap for situations which would cause the processor to suspend execution if the HCR.TWI bit were clear.

WFE is a mechanism for suspending operation during the polling of a variable (such as a spinlock), and as such can present an opportunity for rescheduling which virtual machine is running. However, as WFE is likely to be of shorter duration than WFI, there may be some situations where WFE is not treated in this way. For this reason as separate trap mechanism is provided for trapping WFE.

When the HCR.TWE bit is set and the core is executing in Non-secure state other than Hyp mode: then the execution of a WFE causes a Hyp Entry Trap for situations which would cause the processor to suspend execution if the HCR.TWE bit were clear.

8.12 Access to Jazelle-DBX functionality

A hypervisor trap is introduced to trap use of the Jazelle-DBX functionality. This trap covers:

- Any access to the JOSCR, JMCR or SUB-ARCHITECTURE DEFINED configurations registers which are not described as UNDEFINED in [1]. **Note:** The JIDR is explicitly excluded from this list
- Any attempt to execute an instruction with J=1, T=0 that is not UNDEFINED for other reasons
- Any attempt to execute a BXJ instruction.

These traps apply for all implementations of Jazelle, including trivial implementations.

When the HSTR.TJDBX bit is set, any of these cases in Non-Secure state other than Hyp mode causes a Hyp Entry Trap.

8.13 Access to Thumb-EE Configuration Registers

A hypervisor trap is introduced to trap access to the Thumb-EE configuration registers:

TEECR, TEEHBR

Any attempt to access these registers, if it is not defined as UNDEFINED in [1], from Non-secure state other than Hyp mode, causes a Hyp Entry Trap when the HSTR.TTEE bit is set.

8.14 Using Coprocessor registers

The Hyp Coprocessor Trap Register (HCPTR) provides a facility to define a set of coprocessors (for example Neon or VFP), where an attempt from a Non-secure mode other than Hyp mode to access one of the coprocessors, if the corresponding bit in the HCPTR is '1', causes a Hyp Entry Trap. This approach is closely analogous to the second stage aborts within the memory system, and allows "lazy" VM switching by the hypervisor.

Where access to a coprocessor from a Non-secure mode other than Hyp mode is disabled in the CPACR (indicating an undefined exception to be taken in Non-secure Undefined mode) and it is also trapped in the HCPTR (indicating a Hyp Entry Trap), the Undefined exception to be taken in Non-secure Undefined mode takes priority.

An attempt from Hyp mode to access one of the coprocessors, including the Advanced SIMD Extension, if the corresponding bit in the HCPTR is '1', causes an Undefined exception, which is taken in Hyp mode (see section 4.4.1). The HSR provides syndrome information for this exception. Unlike Hyp mode traps, this applies both when in Debug state as well as when not in Debug state.

> **Note:** This additional use of the HCPTR to trap accesses from within Hyp mode and when in Debug state is provided to allow the HCPTR to be used to trap all access to the Neon and VFP register file for power saving reasons.

There is also a bit in the HCPTR, the TASE bit, that, when '1':

- generates a Hyp Entry Trap when executing any Advanced SIMD instruction from any Non-secure mode other than Hyp mode
- generates an UNDEFINED exception when executing any Advanced SIMD instruction from Hyp mode

This is the equivalent of the CPACR.ASEDIS bit within the CPACR. Where use an Advanced SIMD instruction in a Non-secure mode other than Hyp mode is disabled by the CPACR.ASEDIS bit (indicating an undefined exception to be taken in Non-secure Undefined mode) and it is also trapped in the HCPTR (indicating a Hyp Entry Trap), the Undefined exception to be taken in Non-secure Undefined mode takes priority.

If access to a coprocessor, including the Advanced SIMD extension, is prohibited when in Non-secure state by a bit in the NSACR, the corresponding bits in the HCPTR IGNORE writes from the Non-Secure state, and return the value '1' on reads. In addition the corresponding bits in the HCPTR behave as if they are set to '1' with regard to their effect on Non-secure accesses to the coprocessor.

The top bit of the HCPTR, when '1', generates a Hyp Entry Trap on direct access using an MCR or MRC instruction to the CPACR from Non-Secure Kernel modes. This recognises the fact that the CPACR serves a function as an ID register in identifying whether or not coprocessor functionality is present.

See also section 12.4

8.15 Write Access to the Non-secure Virtual Memory control registers

While the Virtualization Extensions provide facilities to perform the second stage of translation in hardware, having the ability to trap any attempts by the Guest OS to write to the controls of the Non-secure memory system allows the use of paravirtualization approaches for the handling of memory management.

Any attempt that is not UNDEFINED in [1], to write to the following Non-secure registers from Non-secure state other than Hyp mode causes a Hyp Entry Trap when the HCR.TVM bit is set:

SCTLR, TTBR0, TTBR1, TTBCR, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, CONTEXTIDR.

8.16 Generic Trapping of CP15 based functionality

There is a large number of usage models associated with Virtualization, and it is not straightforward to anticipate all such usage models. For this reason, a set of generic traps bits is added which provide traps for each of the “primary” CP15 registers (ie those registers defined by the CRn register field for MCR/MRC instructions, and CRm field for MCRR/MRRC instructions) that have defined or reserved function in [1].

Any attempt to access the registers or functionality for a given CRn/CRm from Non-secure Kernel modes, or from Non-secure User mode if the functionality is not UNDEFINED from Non-secure User mode in [1], causes a Hyp Entry Trap when the corresponding HSTR register bit is set. Where a function using one of these registers is IMPLEMENTATION DEFINED, it is IMPLEMENTATION DEFINED whether Non-Secure User mode accesses are cause this trap or causes an UNDEFINED exception to Non-secure UNDEFINED mode.

These traps can be configured independently for the following CRn/CRm values:

C0, C1, C2, C3, C5, C6, C7, C8, C9, C10, C11, C12, C13, C15

Notes:

- Implementations might provide additional controls in implementation defined registers to provide finer grained control of control of trapping of IMPLEMENTATION DEFINED features.
- The trapping of Non-secure User Mode access to these registers to Hyp mode is expected to be unusual, and should be used where the hypervisor is virtualizing user mode functionality. In most cases, it is expected that Non-secure User Mode access to these registers will result in an UNDEFINED exception taken in Non-secure UNDEFINED mode, as would be the case in the equivalent system without the virtualization extensions.
- The trapping of all attempted accesses to these registers from a Non-secure Kernel Mode overrides the general behaviour described in section 8.1.3

8.17 Trapping General Exceptions to Hyp Mode

A hypervisor trap is introduced to allow the following exceptions when running in Non-secure User mode to be handled in Hyp mode, instead of taking the relevant exceptions in one of the Non-secure Kernel modes:

UNDEFINED Instruction Exceptions

SVC

Alignment Faults other than those derived as a result of the memory type (see section 6.2.11)

Synchronous External Aborts

For UNDEFINED Instruction Exceptions, Alignment Faults and Synchronous External Aborts, these exceptions are taken as Hyp Entry Traps when the HCR.TGE bit is set.

If an UNDEF instruction is taken as a Hyp Entry Trap because of this mechanism, it is reported as an UNKNOWN REASON in the HSR.

Executing in a Non-secure Kernel mode with this trap set is UNPREDICTABLE.

Executing in Non-secure User mode with SCTLR.M set with this trap set is UNPREDICTABLE.

In addition, if an SVC instruction is executed in a Non-secure User mode when this mechanism is enabled, then it behaves in the same way as an HVC instruction, causing an entry into Hyp mode. An SVC is distinguished from an HVC in the HSR. If SVC is unconditional, the immediate field in the HSR in this case takes:

A zero-extended 8bit immediate value for the 16-bit Thumb SVC instruction

The bottom16 bits of the immediate for the 32-bit ARM SVC instruction

If SVC is conditional, the immediate field in the HSR in this case is UNKNOWN.

Note: An SVC instruction is not trapped as a result of TGE, but instead the exception is routed to Hyp mode. This means that the preferred exception return for the SVC when it is routed to Hyp mode is the instruction after the SVC instruction, as opposed to the SVC instruction itself, which would have been the case if the SVC instruction had been trapped.

If an Alignment Fault or Synchronous External abort caused by a data access is taken as a Hyp Entry Trap because of this mechanism, the HSR records this as a Data Abort that caused an entry into Hyp mode, and the associated syndrome information is presented.

If a Synchronous External abort caused by an instruction access is taken as a Hyp Entry Trap because of this mechanism, the HSR records this as a Prefetch Abort that caused an entry into Hyp mode, and the associated syndrome information is presented.

Conditional Alignment Faults, Synchronous External Data aborts and SVC instructions are only trapped by this mechanism if the instruction that causes the exception passes its condition codes.

This mechanism is enabled when the HCR.TGE bit is set.

8.18 Debug and Trace Traps

See Section 12 *Debug* on page 52.

9 OTHER VIRTUALIZATION SUPPORT

9.1 Upgrading the Shareability of barriers

The ARM architecture provides qualifiers to the DMB and DSB barriers to allow the definition of the required shareability domain in which the barriers operate. In a virtualized system, facilities are provided which allow the shareability attributes of the processor to be increased as part of the second stage translation. The hypervisor is able to upgrade the shareability of barriers in a manner that is consistent with this, using a two bit field in the HCR. The HCR.BSU field upgrades the required shareability domain of the DMB and DSB barriers for barriers issued when in Non-secure state and not in Hyp mode, by setting the minimum shareability domain that is applied to any barrier.

00 – No Effect

01 – Inner Shareable

10 – Outer Shareable

11 – Full System

This value is combined with the specified level of the barrier held in its instruction, according to the algorithm for combining Shareability attributes described in section 6.2.3.1.

9.2 Trace and Performance Monitor extensions

See Section 12 *Debug* on page 52.

9.3 Provision of a Software Thread ID Register for Hyp mode

A dedicated Software Thread ID register Is provided for use by a hypervisor running in Hyp mode (the HTPIDR). It is a 32-bit register which is Read/Write in Hyp mode and in Secure Monitor mode (with SCR.NS==1). It is not accessible in Non-secure Kernel modes. It is provided to allow the storage of thread identifying information, in same way as the other Software Thread ID registers described in [1, section B3.12.46]

In keeping with the other Software Thread ID registers, it is never updated by hardware.

10 ADDITIONAL SECURITY FACILITIES

10.1 Overview

The ARM Security Extensions provide facilities to aid the creation of secure systems with isolated, high trusted code running in the Secure state. In the light of feedback from Security vendors, there are a small number of additional features that would improve the resilience of such code to “catastrophic” failure in the event of an error in the code. These are added as part of the Virtualization Extensions as configurable additional features.

10.2 Restriction on Secure Instruction Fetch

The ARM Security Extensions provide a capability that, when executing in Secure state, virtual addresses can be translated to physical addresses in either the secure or Non-secure physical address space. This is a necessary requirement to allow the exchange of data between the Secure and Non-secure state.

The encoding of this mechanism (as an additional bit in the translation tables) has the consequence that it also allows the Secure state instruction fetch to fetch instructions from Non-secure physical memory. Most uses of the Security Extensions do not require that capability, and it has been argued that the ability to fetch instructions for execution in the Secure state from Non-secure physical memory could be seen as a “single point of failure” in Secure code.

For this reason, the Virtualization Extensions add a configuration bit (SIF – secure instruction fetch) bit is added into the SCR which causes any attempt to execute instructions from the Secure state from virtual addresses where the translation accesses the Non-secure physical memory map to give a Permission fault. In keeping with other permission faults, this only applies to Client domains.

The Secure Instruction Fetch bit is expected to be static during normal operation. In particular, it is IMPLEMENTATION DEFINED whether the TLB caches the effect of the SIF bit.

The generic sequence to ensure the synchronization of a change to the SIF bit is:

```
Change the SCR.SIF bit
ISB ; This ensures synchronization of the change
Invalidate entire TLB
DSB ; This completes the TLB Invalidation
ISB; this ensures instruction synchronization
```

10.3 Prevention of Execution from Writeable locations

A common contributory factor in security problems is where execution is diverted to run in a region of memory containing data. XN (Execute Never) functionality within the translation tables was introduced into the ARM architecture to act to make such functionality impossible. Clearly this requires the software discipline that translation tables are written to set XN whenever a region is writeable.

In some systems, and in particular those with a focus on security, it is a design principle that a region of memory that is writeable will never be executed from, without a change to the translation table entry for that memory. In such a system, the forcing of XN whenever a page is writeable creates an additional level of protection, by removing the possibility of a corrupt translation being able to create a writeable page.

For this reason, the Virtualization Extensions introduce a new banked bit into the SCTLR, and also into the HSCTLR called the WXN bit. This bit ensures that instruction fetches to Client regions which are writeable at stage 1 of the memory translation are treated as XN regardless of the setting of the actual XN bit. This only applies when the stage 1 of the memory translation is enabled.

Some operating systems will not be able to support using this bit. A further banked control bit is added to the SCTLR, the UWXN bit. In the Secure state, this bit ensures that Privileged mode instruction fetches to Client regions which are User Writeable are treated as XN, regardless of the actual setting of the XN bit. In the Non-secure state, this bit ensures that Kernel mode instruction fetches to Client regions which are User Writeable at stage 1 of the memory translation are treated as XN, regardless of the actual setting of the XN bit. This only applies when the stage 1 of the memory translation is enabled.

These bits are expected to be static for a given virtual machine during normal operation. In particular, it is IMPLEMENTATION DEFINED whether the TLB caches the effect of these bits associated with a particular VMID.

The generic sequence to ensure the synchronization of a change to these bits within a given virtual machine is:

```
Change the associated bit
ISB ; This ensures synchronization of the change
Invalidate entire TLB of associated entries
DSB ; This completes the TLB Invalidation
ISB; this ensures instruction synchronization
```

10.4 Prevention of Root Kits using Hyp Mode or the Secure state

A well-known security attack for systems with unused deep levels of privilege is that malicious software install themselves in those levels of privilege and so can lie undetected by the operating system running in the Kernel and Non-Privileged Non-secure modes. The approach is typically used by *Rootkits*. The Virtualization Extensions offer two approaches to help prevent this.

- Non-secure Hyp mode is disabled by default, and must be enabled by setting the SCR.HCE bit. When the SCR.HCE bit is 0:
    - The HVC instruction is UNDEFINED when executed in Non-secure Kernel modes.
    - The HVC instruction is UNPREDICTABLE when executed from Hyp mode

  Note: this is the only effect of the SCR.HCE bit.
- Entry into Secure state can be disabled (other than at reset) by setting a new bit, the SCR.SCD bit. When the SCR.SCD bit is 1:
    - The SMC instruction is UNDEFINED unless trapped by the HCR.TSC mechanism (see section 8.10) when executed from the non-secure state
    - The SMC instruction is UNPREDICTABLE when executed from the secure privileged modes.

Note: These bits, combined with the reset values of the controls for routing exceptions to either Hyp mode or Monitor mode, prevent entry into these modes by any exceptions. The effect of these controls on the call instructions in the modes that they are protecting entry to is UNPREDICTABLE as it is not expected that these instructions will be executed in those modes with the controls at the values that cause UNPREDICTABLE behaviour.

11 ADDITIONAL FEATURES

11.1 SWP/SWPB is Optional

For implementations with the Virtualization Extensions, the inclusion of SWP and SWPB in the ARM instruction set is optional. This follows on as a progression from its deprecation as part of ARMv6.

If an implementation does not support the SWP and SWPB instructions, the ID information in the ID_ISAR0 (Swap_instrs) and ID_ISAR4 (SWP_frac) indicate that SWP and SWPB are not supported. In addition, in such implementations it is not possible for the SCTLR.SW bit to be set; instead this bit behaves as RAZ/WI.

11.2 Deprecated Features

The NSACR.RFR bit is deprecated by the Virtualization Extensions

The SCTLR.VE bit is deprecated by the Virtualization Extensions

The SCTLR.HA bit is deprecated by the Virtualization Extensions

12 DEBUG, TRACE AND PERFORMANCE MONITORS

12.1 Introduction

The Virtualization Extensions build on other extensions to the debug, trace and performance monitor components of the ARM architecture.

- Performance Monitors version 2, described in [4], extends the PMU to support filtering of event counting by processor mode. [5] describes the impact of Virtualization on the PMU architecture.
- Changes to the trace architecture to support Virtualization are described separately.
- v7.1 Debug simplifies and rationalizes the debug architecture. The details will be published in the revision C release of [1]. The impact of Virtualization on the Debug architecture is described in this chapter.

Each of these extensions is a pre-requisite for the Virtualization Extensions.

12.2 Virtualization support based on Monitor debug-mode

The ARMv7 Architecture has a Monitor debug-mode to handle debug events which are treated as debug exceptions. Those debug exceptions associated with instruction fetch (breakpoints, the BKPT instruction and vector traps) are taken as Prefetch Aborts, while those associated with data accesses (watchpoints) are taken as Data Aborts.

The Virtualization Extensions provide a control bit in the Hyp Debug Control Register (HDCR), the HDCR.TDE bit, which makes all Debug Exceptions be taken in Hyp Mode, using the vector offset 0x14 as described in section 4.4.

Monitor debug-mode exceptions are not permitted in Hyp mode (see sections 12.7.1, 12.9 and 12.10).

12.3 Trapping Access to CP14 Debug Registers

12.3.1 HDCR.TDA

A bit, the HDCR.TDA bit, is added to the HDCR as part of the Virtualization Extensions, which causes the generation of a Hyp Entry Trap for any access from a Non-secure mode other than Hyp mode to the Baseline CP14 registers and the Extended CP14 registers other than the following registers:

- DBGDRAR and DBGDSAR.
- DBGOSLSR, DBGOSLAR
- DBGOSDLR
- DBGPRCR
- Any IMPLEMENTATION DEFINED integration registers.
- Any IMPLEMENTATION DEFINED that are trapped as a result of the HDCR.TDOSA (see section 12.3.2)

The registers trapped by the HCDR.TDA bit include the cases of an STC from DBGDTRRXint and LDC to DBGDTRTXint.

This trap acts as a “second stage” of exception, and so is only required to apply to accesses to Debug CP14 registers which are not UNPREDICTABLE and which do not cause an UNDEFINED exception. See tables C6-8, C6-10 and C6-11 of [1] for more detail.

Note: The exception is permitted, but not required, to apply to accesses to Debug CP14 registers which are UNPREDICTABLE in those tables, when the HDCR.TDA bit is set.

Accesses to some CP14 Debug registers have side-effects; where those accesses have been trapped by this new mechanism, the side-effects do not occur.

This exception does not occur when in Debug State.

12.3.2 HDCR.TDOSA

A bit, the HDCR.TDOSA bit, is added to the HDCR as part of the Virtualization Extensions, which causes the generation of a Hyp Entry Trap for any access from a Non-secure mode other than Hyp mode to generate a Hyp Entry Trap to the following registers:

- DBGOSLSR, DBGOSLAR
- DBGOSDLR
- DBGPRCR
- Any IMPLEMENTATION DEFINED integration registers.
- Any IMPLEMENTATION DEFINED that the implementation with similar functionality.

Accesses to some CP14 Debug registers have side-effects; where those accesses have been trapped by this new mechanism, the side-effects do not occur.

This exception does not occur when in Debug State.

12.4 Trapping Access to CP14 Trace Registers

A bit, the HCPTR.TTA bit, is added to the HCPTR as part of the Virtualization Extensions, which causes any access to the CP14 Trace registers from a Non-secure mode other than Hyp mode to generate a Hyp Entry Trap, and access to the CP14 Trace Registers from Hyp mode to generate an UNDEFINED exception.

The Hyp Entry Trap acts as a “second stage” of exception, and so is only required to apply to accesses to CP14 Trace registers which are not UNPREDICTABLE and which do not cause an UNDEFINED exception, as described in the appropriate trace extension manual.

Note: The exception is permitted, but not required, to apply to accesses to Trace CP14 registers which are UNPREDICTABLE in the trace extension manual, when the HCPTR.TTA bit is set.

Accesses to some CP14 Trace registers have side-effects; where those accesses have been trapped by this new mechanism, the side-effects do not occur.

The Hyp Entry Trap generated by this mechanism does not occur when in Debug State. The UNDEFINED exception in Hyp mode generated by this mechanism does occur in Debug State.

Note: this mechanism is provided to allow trapping to be used to prevent access to the ETM CP14 registers when it is powered down, in a similar way as is provided for in other coprocessors.

12.5 Trapping Access to CP15 Performance Monitor Registers

See Section 8.9 *Accesses to Performance Monitor Functionality* on page 45.

12.6 Trapping Access to CP14 Debug ROM registers

A bit, the HDCR.TDRA bit, is added to the HDCR as part of the Virtualization Extensions, which causes any access to the DBGDRAR and DBGDSAR from a Non-secure mode other than Hyp mode to generate a Hyp Entry Trap. This applies to accesses using either MRC or MRRC instructions.

This exception does not occur when in Debug State.

12.7 Control of Breakpoint and Watchpoint matching

12.7.1 Breakpoint and Watchpoint Matching when in Hyp Mode

The DBGBCR and DBGWCR registers contain fields (bits[15:14] and bits[2:1]) that determine whether matching can occur in Secure and/or Non-secure state, and in particular modes. The bit[15:14] field is extended to provide control to allow Breakpoint and Watchpoint matching when in Hyp mode.

Breakpoints and Watchpoints events can only occur when in Hyp mode for Halting debug-mode. If Monitor debug-mode is selected, no Breakpoint or Watchpoint events can occur in Hyp mode.

12.7.2 BRP extensions for VMID matching

A new 8-bit register (the Debug Breakpoint Extended Value Register) is added to BRP which support Context ID comparison to allow them to also optionally include a comparison with the current VMID as part of the linked matches.

The selection of the VMID as part of the linked match is controlled by fields in the associated DBGBCR.

12.8 Distinguishing Stage 1 and Stage 2 Data Aborts in Debug State

When a Data Abort exception is generated by a synchronous data abort in Debug state, the behaviours described in section C5.7 are followed, but are extended as follows:

- The updating of the DFAR and DFSR described in that section are extended to follow the rules described in sections 6.2.7.1 and 6.2.7.3 associated with the additional syndrome information introduced as part of the Virtualization Extensions.
- In order to provide a mechanism to allow the debugger to distinguish whether an abort was a first stage fault or a second stage fault, a new bit (the FS bit) is added to the DBGDSCR. This bit is used in addition to the SDABORT_l bit.
- DBGDSCR.FS is not sticky; it is not cleared by writes to DBGDRCR.ClearStickyExceptionFlags, and it does not affect the DBGITR. DBGDSCR.FS is set on every synchronous Data Abort taken in Debug state.
- On a second stage fault, the architectural fault status information is presented in the HSR, any implementation defined auxiliary fault status information is presented in the HADFSR, and architectural fault address information is presented in the HDFAR or HPFAR as appropriate

12.9 Debug Vector Catch Register

New Vector Catch fields as added to allow Hyp mode vector entries to be caught, using bits [23:17], following the pattern of the other vector catch fields. These vector entries only match when the instruction memory accesses are made in Hyp mode.

Monitor debug-mode exceptions are not permitted in Hyp Mode, and therefore these catches do not occur when Monitor debug-mode is selected.

With Virtualization Extensions, if HDCR.TDE == 1 and the processor is in Non-secure state, meaning debug exceptions are trapped into Hyp mode, Prefetch and Data Abort vector catches are not UNPREDICTABLE when Monitor debug-mode is selected. Instead they must generate a debug exception trapped into Hyp mode. This is in contrast to the situation before the Virtualizations Extensions, described in [1, section C3.2.6] which describes that Prefetch and Data Abort vector catches are UNPREDICTABLE when Monitor debug-mode is selected.

12.10 Monitor Debug-mode use of BKPT instruction in Hyp Mode

Monitor debug-mode exceptions are not permitted in Hyp mode. In keeping with all other situations where a Monitor debug-mode BKPT instruction is not permitted, if a BKPT instruction is executed in Hyp mode when

Monitor debug-mode is enabled, then this generates a Prefetch abort. This Prefetch abort will be taken in Hyp mode.

12.11 Sample Based Profiling

The Debug Virtualization ID Sampling Register (DBGVIDSR) is part of the Sample Based profiling support. The Virtualization Sampling Register samples the VMID and NS state whenever the Program Counter Sampling Register, DBGPCSR, samples the Program Counter. This enables a debugger to associate a Program Counter sample with the Virtual machine running on the processor.

DBGVIDSR is introduced as part of v7.1 Debug, although without Virtualization Extensions it only samples the NS state; for more information.

12.12 Performance Monitors

Fine-grained control is provided for the performance counters so that for each counter, it is possible to define whether it can count in each of the following modes (including certain combinations of these modes):

Non-secure state:

Non-privileged mode

Kernel modes

Hyp mode

Secure state

Non-privileged mode

Privileged modes

This is controlled by bits within the individual counters control registers.

In addition, a facility is provided to allow the set of monitors accessible by a Guest operating system to be smaller than the full set of monitors implemented by the hardware. This effectively reserves some counters for use by the hypervisor.

The number of monitors that are presented to the GuestOS is configurable in a field in the HDCR, the HDCR.HPMN. This field must be set to be less than or equal to the value PMCR.N (or else the behaviour of the counters is UNPREDICTABLE), and represents the value of the PMCR.N field when the PMCR is read from Non-secure Kernel modes.

Performance counters whose number lies in the range:

HDCR.HPMN <= Counter Number < PMCR.N

cannot be accessed from Non-secure Kernel modes – setting the PMSELR.SEL field to such a value is UNPREDICTABLE, but must not permit Non-secure Kernel mode access to the registers associated with that counter. The enabl*i*ng of these counters is controlled by the HDCR.HPME bit.

In addition, the fields in the following registers corresponding to the Counter Numbers in this range are RAZ/WI when accessed from Non-secure Kernel modes.

PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMINTENSET, PMINTENCLR

The detailed proposals for handling the performance counters are beyond the scope of this document.

12.13 Trace

The ETM and PFT architectures ([6] and [7], respectively) allow filtering of trace by NS state, and indication of the NS state in the trace stream.

The extensions for Virtualization allow:

- Filtering of trace by combinations of mode:

  Non-secure state:

Non-privileged mode

Kernel modes

Hyp mode

Secure state

Non-privileged mode

Privileged modes

- Indication of Hyp mode in the trace stream.

13 STATE ADDED BY THE VIRTUALIZATION EXTENSIONS

13.1 General Behaviours

The registers listed in this section are all UNKNOWN at reset unless otherwise stated.

13.2 Hyp Configuration Register (HCR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is reset to 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNK/ SBZP				TGE	TVM	TTLB	TPU	TPC	TSW	TAC	TIDCP	TSC	TID3	TID2	TID1	TID0	TWE	TWI	DC	BSU [1:0]		FB	VA	VI	VF	AMO	IMO	FMO	PTW	SWIO	VM

TGE(Bit[27]): Trap General Exceptions (see section 8.17)

TVM(Bit[26]): Trap Virtual Memory Controls (see section 8.15)

TTLB (Bit[25]): Trap TLB maintenance instructions (see section 8.7)

TPU (Bit[24]): Trap Cache maintenance instructions to point of unification (see section 8.6)

TPC (Bit[23]): Trap Data/Unified cache maintenance instructions to point of coherency (see section 8.5)

TSW (Bit[22]: Trap Data/Unified cache Set/Way instructions (see section 8.4)

TAC (Bit[21]): Trap Auxiliary Control Register Accesses (see section 8.8)

TIDCP (Bit[20]): Trap Lockdown (see section 8.3)

TSC(Bit[19]): Trap SMC (see section 8.10)

TID3 (Bit[18]): Trap ID Group 3 (see section 8.2.2.6)

TID2 (Bit[17]): Trap ID Group 2 (see section 8.2.2.5)

TID1 (Bit[16]): Trap ID Group 1 (see section 8.2.2.4)

TID0 (Bit[15]): Trap ID Group 0 (see section 8.2.2.2)

TWE (Bit[14]): Trap WFE (see section 8.11)

TWI (Bit[13]): Trap WFI (see section 8.11)

DC (Bit[12]): Default Cacheable (see section 0)

BSU (Bit[11:10]:Barrier Shareability Upgrade (see section 9.1)

FB (Bit[9]): Force Broadcast of TLB maintenance, BPIALL and ICIALLU instructions (see sections 6.2.8.6 and 6.2.8.8.1)

VA (Bit[8]) : Virtual External Asynchronous Abort (see section 5.2)

VI (Bit[7]): Virtual IRQ interrupt (see section 5.2)

VF (Bit[6]): Virtual FIQ interrupt (see section 5.2)

AMO (Bit[5]): A-bit Mask Override (see section 5.1)

IMO (Bit[4]): I-bit Mask Override (see section 5.1)

FMO (Bit[3]): F-bit Mask Override (see section 5.1)

PTW (Bit[2]): Protected Table Walk (see section 6.2.9)

SWIO (Bit[1]): Set/Way Invalidation Override (see section 6.2.8.4)

VM (Bit[0]): Second Stage of Translation Enable (see section 6.2.8.1.1)

13.3 Hyp Debug Control Register (HDCR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

This register is reset so that bits[31:8, 6:5] are 0. Bit[7] is UNKNOWN at reset. Bits[4:0] are reset the value of PMCR.N.

31	12	11	10	9	8	7	6	5	4	0
UNK/SBZP		TDRA	TDOSA	TDA	TDE	HPME	TPM	TPMCR	HPMN	

TDRA (Bit[11]): Trap Debug ROM access (see section12.6)

TDOSA (Bit[10]): Trap Debug OS related register Access (see section 12.3.2)

TDA (Bit[9]): Trap Debug Access (see section 12.3.1)

TDE (Bit[8]): Trap Debug Exceptions (see section 12.2)

HPME (Bit[7]): Hypervisor Performance Monitor enable override bit (see section 12.12)

TPM(Bit[6]): Trap Performance Monitors (see section 8.9)

TPMCR(Bit[5]): Trap Performance Monitor Control Register (PMCR), see section 8.9

HPMN(Bits[4:0]): Performance Monitor count. Specifies the number of performance counters that can be accessed by the Guest OS.
See section 12.12

More information on HDCR[7:0] will be provided as part of the performance monitor specification.

13.4 Hyp Coprocessor Trap Register (HCPTR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

Bits in this register which are implemented are reset to 0.

31	30	21	20	19	16	15	14	13		0
TCPAC	UNK/SBZP		TTA	UNK/SBZP		TASE	(0)	TCP13 to TCP0		

TCPAC (Bit[31]): Trap direct access to CPACR from Non-secure Kernel modes to Hyp mode. See section 8.14

TTA (Bit[20]): Trap Trace Access (see section 12.4)

TASE (Bit[15]): Trap Advanced SIMD Extension usage from the Non-secure state to Hyp mode. See section 8.14

On an implementation that:

- Implements VFP and does not implement Advanced SIMD, this bit is RAO/WI.
- Does not implement VFP or Advanced SIMD, this bit is RAO/WI.
- Implements both VFP and Advanced SIMD, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

TCP<n> (Bit[n]): Trap use of CP<n> from the Non-secure modes to Hyp mode. See section 8.14

For bits that correspond to coprocessors that are not implemented, it is IMPLEMENTATION DEFINED whether the bits:

- behave as RAO/WI
- can be written by Hyp mode and Secure Monitor mode (with SCR.NS==1).

If more than one coprocessor is used to provide a set of functionality then having different values for the HCPTR fields for those coprocessors can lead to UNPREDICTABLE behaviour. This applies to the CP10 and CP11 for VFP.

See section 8.14

13.5 Hyp System Trap Register (HSTR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

This register is reset to 0.

31	18	17	16	15	14	13	5	4	3	0
UNK/SBZP		TJDBX	TTEE	T15	UNK/SBZP	T13-T5		UNK/SBZP	T3 to T0	

TJDBX(Bit[17]) Trap Jazelle DBX (see section 8.12)

TTEE(Bit[16]): Trap ThumbEE (see section 8.13)

T<n> (Bit[n]): Trap to Hyp mode Non-secure priv (and defined Non-secure user) use of

- CP15 CRn == C<n> for MCR or MRC instructions
- CP15 CRm == C<n> for MCRR or MRRC instructions

See section 8.16

13.6 Hyp IPA Fault Address Register (HPFAR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is UNKNOWN at reset, and is made UNKNOWN when executing in Non-Secure modes other than Hyp mode.

31	4	3	0
FIPA[39:12]		UNK/SBZP	

FIPA (Bits[31:4]): Faulting IPA bits [39:12].

13.7 Hyp Syndrome Register (HSR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. It holds syndrome information for exceptions taken in Hyp mode. This register is UNKNOWN at reset, and made UNKNOWN when executing in Non-Secure modes other than Hyp mode.

31	26	25	24	0
EC		IL	ISS	

EC (bits[31:26]): Exception Class field for the Hyp exception:

- 0x00: Unknown reason – all other fields invalid. This value is used in all cases that are not covered by some other Exception class below.
- 0x01: Trapped WFI/WFE
- 0x03: Trapped MCR/MRC to CP15
- 0x04: Trapped MCRR/MRRC to CP15
- 0x05: Trapped MCR/MRC to CP14
- 0x06: Trapped LDC/STC to CP14 (DBGDTRTXint /DBGDTRRXint only)
- 0x07: Trapped Coprocessor Usage to CP0-CP13 or Advanced SIMD Extensions due to HCPTR
- 0x08: Trapped MRC (or VMRS) to CP10 for MVFR0, MVFR1 or FPSID for ID Group traps unless trapped by the HCPTR
- 0x09: Trapped Jazelle instruction : trapping of CPSR.{J,T} = {1,0}
- 0x0A: Trapped BXJ
- 0x0C: Trapped MRRC to CP14
- 0x11: SVC which is taken in Hyp mode.
- 0x12: HVC
- 0x13: Trapped SMC
- 0x20: Instruction Abort that caused an entry into Hyp mode (Second stage abort, External/Parity Fault on a second stage translation table walk, ebug event or External/Parity faults routed to Hyp mode as a result of HCR.TGE)
- 0x21: Instruction Abort that was taken while executing within Hyp mode (First stage abort, External/Parity Fault or debug event)
- 0x24: Data Abort that caused an entry into Hyp mode (Second stage abort, External/Parity Fault or debug event, alignment fault)
- 0x25: Data Abort that was taken while executing within Hyp mode (First stage abort, External/Parity Fault or debug event, alignment fault)
- Others: Reserved

IL (bit[25]): Instruction Length. This field indicates the size of the instruction that has been trapped

- 0: 16-bit
- 1: 32-bit

This field is not valid for Instruction Aborts, or for Data Aborts that do not have instruction syndrome information or if the instruction syndrome is not valid. In this case, the field is UNK/SBZP

ISS (bits[24:0]): Instruction Specific Syndrome. This field contains syndrome information for each class of instruction which is trapped (as described by the IC field). These are described in more detail in section 13.7.1

13.7.1 HSR Class specific use of ISS[24:0]

13.7.1.1 Use of ISS[24:20] when HSR[31:30] == '00'

When HSR[31:30] == '00', for all uses of the HSR other than the unknown reason code, the field ISS[24:20] takes the following meaning:

ISS[24]: CV, COND Valid. Indicates that the COND field of the instruction is valid or not

0: Not Valid

1: Valid

If this field is 0, then the conditionality has to be determined from the SPSR.IT bits

Note: ARM instructions are presented with CV==1.

Unconditional Thumb or ThumbEE instructions, or conditional Thumb or ThumbEE instructions which are known to pass their condition codes can be presented as either:

CV==1, COND = 0xE

CV ==0.

ISS[23:20]: Condition field. This field is only valid if CV==1. If CV==0, then this field in UNK/SBZP.

Note: It is IMPLEMENTATION DEFINED under some circumstances whether a conditional instruction that fails its condition code generates an UNDEFINED Exception. If an Implementation only takes the exception when the condition passes, then it is permissible that the syndrome register denotes those instructions as Unconditional (0xE), even if they actually were conditional.

Note: This use of ISS[24:20] exists because the architecture permits an implementation to cause Hyp Traps for instructions that fail their condition codes, as described in section 8.1.1

Except as described above, all unconditional instructions use the value 0xE as their condition.

13.7.1.2 General use of ISS field

The ISS field is used for different purposes, depending on the class of the exception that caused the HSR to be written.

WFI/WFE

ISS[24:20]: See section 13.7.1.1

ISS [19:1]: UNK/SBZP

ISS[0]: 0: WFI

1: WFE

MCR/MRC to CP15, CP14 or CP10

ISS[24:20]: See section 13.7.1.1

ISS[19:17] Op2

ISS[16:14] Op1

ISS[13:10] CRn

ISS[9] UNK/SBZP

ISS[8:5] Rt

ISS[4:1] CRm

ISS[0]: Direction:

0: MCR

1: MRC

MCRR/MRRC to CP15 or to CP14:

ISS[24:20]: See section 13.7.1.1

ISS[19:16] Op1

ISS[15:14] UNK/SBZP

ISS[13:10] Rt2

ISS[9] UNK/SBZP

ISS[8:5] Rt1

ISS[4:1] CRm

ISS[0]: Direction:

0: MCRR

1: MRRC

LDC/STC to CP14:

ISS[24:20]: See section 13.7.1.1

ISS[19:12] imm8

ISS[11:9] UNK/SBZP

ISS[8:5] If ISS[3] == 0: Rn

If ISS[3] == 1: UNKNOWN

ISS[4] 1: Add Offset; 0: Subtract offset

This corresponds to the U bit of the LDC/STC addressing mode

ISS[3:1]: Addressing mode;

000: Immediate Unindexed

001: Immediate Post-Indexed

010: Immediate Offset

011: Immediate Pre-indexed

100: Literal Unindexed (LDC only, ARM only) – Reserved for STC.

101: Reserved

110: Literal (LDC only) – Reserved for STC

111: Reserved

The ISS[2:1] fields correspond to the P and W bits of the LDC/STC addressing mode

ISS[0]: Direction (and Register):

0: STC (DBGDTRRXint)

1: LDC (DBGDTRTXint)

Coprocessor Access to CP0-CP13 or Advanced SIMD usage

ISS[24:20]: See section 13.7.1.1

ISS[19:6] UNK/SBZP

ISS[5]: Trapped Advanced SIMD usage; this is set for all trapped Advanced SIMD instructions that are not also VFP instructions as a result of any traps in the HCPTR.

ISS[4]: UNK/SBZP

ISS[3:0] CP Number. This field is only valid if ISS[5] == 0 otherwise it is UNK/SBZP. This field gives the value 0xA when valid for VFP instructions.

Jazelle Instruction

ISS[24:20]: See section 13.7.1.1

ISS[19:0]: UNK/SBZP

BXJ Instruction

ISS[24:20]: See section 13.7.1.1

ISS[19:4]: UNK/SBZP

ISS[3:0]: Rm

HVC or SVC

ISS[24:16]: UNK/SBZP

ISS[15:0]: imm16

Note: SVC instructions cannot be trapped if they fail their condition code, and so do not use the condition code mechanism described in section 13.7.1.1

SMC

ISS [24:0]: UNK/SBZP

Note: SMC instructions cannot be trapped if they fail their condition code, and so do not use the condition code mechanism described in section 13.7.1.1

Instruction Aborts (both types)

ISS[24:10]: UNK/SBZP

ISS[9]: EA, External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

Note: This is the ExT field, defined for the IFSR in [1, section B3.12.28]

ISS[8]: UNK/SBZP

ISS[7]: S1PTW, Indicates the instruction fault came from a second stage fault during a first stage translation table walk

Note: applies to Stage 2 faults only – UNK/SBZP otherwise

ISS[6]: UNK/SBZP

ISS[5:0]: IFSC, Instruction Fault Status Code defined in [2]

Note: The Domain faults are not possible, and the associated encodings are Reserved.

Data Aborts (both types)

ISS(bit[24]): ISV, Instruction Syndrome Valid: Indicates whether the faulting instruction was of a class the emulation of which can be accelerated using the information contained within the other fields of the syndrome associated with the instruction (bits[24:16] of the ISS).

This bit is 0 for aborts other than those generated by the second stage of translation, excluding second stage aborts on first stage translation table walks.

ISS(bits[23:22]): SAS, Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

‘00’: Byte

‘01’: Halfword

’10’: Word

‘11’ : Reserved

This field is only valid when ISS[24]==1, otherwise it is UNK/SBZP.

ISSbit[21]) SSE, Syndrome Sign Extend. Indicates whether the data item should be sign extended for load operations. This is only applicable to byte and halfword accesses (it is 0 otherwise).

This field is only valid when ISS[24]==1, otherwise it is UNK/SBZP.

ISSbit[20] UNK/SBZP

ISSbit[19:16]: SRT, Syndrome Register transfer. The register number of the Rt operand of the faulting instruction. This is the destination register for loads, the source register for stores. In general must be combined with the Mode saved in the SPSR in order to determine the physical register to be accessed.

This field is only valid when ISS[24]==1, otherwise it is UNK/SBZP.

ISSbit[15:10] UNK/SBZP

ISS[9]: EA, External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

Note: This is the ExT field, defined for the DFSR in [1, section B3.12.28]

ISS[8]: CM, Indicates the data fault came from a Cache Maintenance Operation or VA to PA operations (see section 6.2.8.10.2) for synchronous faults only

ISS[7]: S1PTW Indicates the data fault was from a second stage fault during a first stage translation table walk

Note: applies to Stage 2 faults only – UNK/SBZP otherwise

ISS[6]: WnR,Write not Read bit. Indicates whether the abort was caused by a write or a read access:

0 Abort caused by a read access

1 Abort caused by a write access.

For faults on CP15 cache maintenance operations, including the VA to PA translation operations, this bit always returns a value of 1.

ISS[5:0]: DFSC, Data Fault Status Code, defined in [2]

Note: The Domain faults are not possible, and the associated encodings are Reserved.

13.8 Hyp Auxiliary Data Fault Status Syndrome Register (HADFSR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is UNKNOWN at reset, and its contents are IMPLEMENTATION DEFINED. An implementation that does not use this register might implement its contents as UNK/SBZP.

31		0
IMPLEMENTATION DEFINED		

13.9 Hyp Auxiliary Instruction Fault Status Syndrome Register (HAIFSR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is UNKNOWN at reset, and its contents are IMPLEMENTATION DEFINED. An implementation that does not use this register might implement its contents as UNK/SBZP.

31		0
IMPLEMENTATION DEFINED		

13.10 Hyp Translation Table Base Register (HTTBR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions The format of this is described in the description of the new translation system Translation Table Base Register in [2].

This register is UNKNOWN at reset.

13.11 Hyp Translation Control Register (HTCR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

The format of this is described in the description of the new translation system First Stage Translation Table Base Control Register in [2].

This register is UNKNOWN at reset.

13.12 Hyp Memory Attribute Indirection Registers (HMAIR0/1)

These are two a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

The format of these registers is described in the description of the new translation system section describing the MAIR0/HMAIR0 and MAIR1/HMAIR1 in [2].

This register is UNKNOWN at reset.

13.13 Virtualization Translation Table Base Register (VTTBR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions The format of this is described in the description of the new translation system Translation Table Base Register in [2].

This register is UNKNOWN at reset, except for the VMID field which is reset to 0.

13.14 Virtualization Translation Control Register (VTCR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions

The format of this is described in the description of the new translation system Second Stage Translation Table Base Control Register in [2].

This register is UNKNOWN at reset.

13.15 Virtualization Processor ID Register (VPIDR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is reset to the MIDR value.

31		0
VPIDR		

VPIDR (bits[31:0]): MIDR value read by Non-secure Kernel accesses to the MIDR

The fields of the VPIDR are the same as the fields in the MIDR, described in [1]

13.16 Virtualization Multiprocessor ID Register (VMPIDR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. This register is reset to the MPIDR value.

31		0
VMPIDR		

VMPIDR (bits[31:0]): MPIDR value read by Non-secure Kernel accesses to the MPIDR

The fields of the VMPIDR are the same as the fields in the MPIDR, described in [1]. Fields in the MPIDR that are UNK, are treated as UNK/SBZP in the VMPIDR.

13.17 Virtualization ID Sampling Register (DBGVIDSR)

This is a RO register added as part of v7.1 Debug. It is only visible on the memory mapped and External debug interfaces. The Virtualization Extensions extend the DBGVIDSR to include the H and VMID fields:

31	30	29	8	7	0
NS	H	UNK		VMID	

VMID (bits[7:0]) VMID sample value, bits [7:0]

The value of the VMID ID Register, VMIDR, bits[7:0], associated with the last PC sample read from DBGPCSR. This field is only valid when the H bit is 0 and the NS bit is 1, otherwise it is UNK.

H (bit[30]) Hyp mode sample value

Indicates whether the last PC sample read from the DBGPCSR was associated with Hyp mode. This field is UNK when NS bit is 0.

NS (bit[31]) NS sample value

Indicates the Secure or Non-secure state associated with the last PC sample read from DBGPCSR

Note: This is the Secure/Non-secure state, not the value of the SCR.NS bit.

The core logic reset value of the DBGVIDSR is UNKNOWN.

13.18 Secure Configuration Register (SCR)

The following bits are added to the Secure Configuration Register as part of the Virtualization Extensions. These bits are reset to 0.

SCD (bit[7]): SMC Disable – causes SMC to be UNDEFINED in Non-secure state. See section 10.4

0 – SMC performs a Secure Monitor Call

1 – SMC is UNDEFINED

HCE (bit[8]): Hyp Call Enable – enabled HVC in Non-secure Kernel Modes. See section 10.4

0 – HVC is UNDEFINED in Non-secure Kernel Modes

1 – HVC is enabled in Non-secure Kernel Modes

SIF (bit[9]) Secure Instruction Fetch. See section 10.2

0 – Secure state instruction fetches are permitted to Non-Secure memory

1 – Secure state instruction fetches are not permitted to Non-Secure memory

13.19 Hyp System Control Register (HSCTLR)

This is a new R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. The Hyp System Control Register performs the same control functions for Hyp mode as the Non-secure SCTLR does for the Kernel and non-Privileged Non-secure modes. Bits that are not banked in the SCTLR are also reflected in this register.

This register is reset to UNK.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	TE	1	1	0	0	EE	0	1	1	FI	0	WXN	1	0	1	0	0	0	I	1	0	0	0	0	1	1	1	1	C	A	M

TE, bit [30] Thumb Exception enable. This bit controls whether exceptions taken in Hyp mode are taken in ARM or Thumb state:

0 Exceptions are handled in ARM state

1 Exceptions are handled in Thumb state.

EE, bit [25] Exception Endianness bit. The value of this bit defines the value of the CPSR.E bit on entry to an exception vector in Hyp mode. This value also indicates the endianness of the translation table data for translation table lookups when executing in Hyp mode , and for second stage translation table lookups in other Non-secure modes.

0 Little endian

1 Big endian.

This is a read/write bit.

FI, bit [21] Fast Interrupts configuration enable bit. This bit can be used to reduce interrupt latency in an implementation by disabling IMPLEMENTATION DEFINED performance features. The permitted values of this bit are:

0 All performance features enabled.

1 Low interrupt latency configuration. Some performance features disabled.

This bit is a read-only bit that reflects the value of the SCTLR.FI bit.

WXN (bit[19]) Write permission implies XN. See section 10.3

0 Hyp Translations with Write Permission are not forced to be XN

1 Hyp Translations with Write Permission are forced to be XN

I (bit[12]) Instruction cache enable bit for memory accesses made in Hyp mode

0 Instruction caches disabled

1 Instruction caches enabled

C (bit[2]) Data/Unified cache enable bit for memory accesses made in Hyp mode

0 Data and Unified caches disabled

1 Data and Unified caches enabled

A (bit[1]) Alignment bit for memory accesses made in Hyp mode

0 Alignment check disabled

1 Alignment check enabled

M (bit[0]): MMU Enable bit for memory accesses made in Hyp mode

0 Disabled

1 Enabled

Note: Hyp mode does not support hardware management of the Access Flag

Note: The HSCTLR does not hold a control for branch prediction, and in general it is assumed that branch prediction will function when executing in Hyp mode. Implementations might choose to implement controls for different types of branch prediction within IMPLEMENTATION DEFINED hardware specific registers.

13.20 System Control Register (SCTLR)

The following bits are added to the Secure and Non-secure System Control Register as banked bits as part of the Virtualization Extensions. These bits are reset to 0.

WXN(bit[19]) Write permission implies XN. See section 10.3

0 Translations with Write Permission are not forced to be XN

1 Translations with Write Permission are forced to be XN

UWXN(bit[20]) User Write permission implies Privileged/Kernel XN. See section 10.3

0 Translations with User Write Permission are not forced to be XN for Privileged/Kernel accesses

1 Translations with User Write Permission are forced to be XN for Privileged/Kernel accesses Translations with Write Permission are XN

13.21 Hyp Vector Base Address Register (HVBAR)

This is a new 32-bit R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions – see section 4.5.2. It has the same format as the MVBAR.

This register is UNKNOWN at reset.

13.22 Hyp Software Thread ID Register (HTPIDR)

This is a new 32-bit R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions – see section 9.3

This register is UNKNOWN at reset.

13.23 Hyp Auxiliary Configuration Register (HACR)

This is a new 32-bit R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. It contains implementation specific controls for trapping implementation specific functionality while executing in Non-secure modes other than Hyp mode. The contents of this register are IMPLEMENTATION DEFINED.

13.24 Hyp Auxiliary Control Register (HACTLR)

This is a new 32-bit R/W register accessible in Hyp mode and Secure Monitor (with SCR.NS==1) only, added for the Virtualization Extensions. It contains implementation specific controls for implementation specific functionality while executing Hyp mode. The contents of this register are IMPLEMENTATION DEFINED.

13.25 Debug Status and Control Register (DBGDSCR)

The following bit is added to the DBGDSCR.

FS (bit [9]): Fault Status bit. This flag is modified by all synchronous Data Abort exceptions that are generated when the processor is in Debug state. The possible values of this bit are:

0 The Data Abort exception was not caused by a second stage fault

1 The Data Abort exception was caused by a second stage fault

This flag is read/write. It is not modified on Data Abort exceptions that are generated in Non-debug state. It is not modified on asynchronous Data Abort exceptions.

See section 12.8

This bit is UNKNOWN at reset.

13.26 Debug Vector Catch Register (DBGVCR)

The following bits are added to the DBGVCR:

Bit[17] HVBAR: Undefined Instruction

Bit[18] HVBAR: HVC

Bit[19] HVBAR: Prefetch Abort

Bit[20] HVBAR: Data Abort

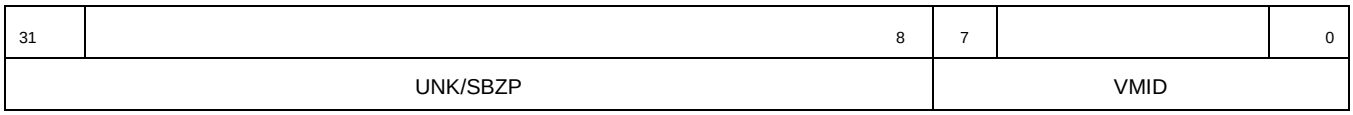
Bit[21] HVBAR: Hyp Entry Exception

Bit[22] HVBAR: IRQ

Bit[23] HVBAR: FIQ

13.27 Debug Breakpoint Extended Value Registers (DBGBXVR)

This is a new set of Common R/W registers added for the Virtualization Extensions for BRPs that support ContextID comparisons. For BRPs that do not support ContextID comparisons, the corresponding register is UNPREDICTABLE.



VMID[7:0]: VMID value for Breakpoint comparison

13.28 Debug Breakpoint Control Registers (DBGBCR)

DBGBCR[13] is the Hyp match field. This determines whether addresses generated in Hyp mode match for Breakpoints:

2 Match cannot occur in Hyp mode

3 Match can occur in Hyp mode

The privileged mode control field (bits[2:1]) in the DBGBCR has no effect for accesses made in Hyp mode.

Only a subset of the settings of the DBGBCR[15:13, 2:1] bits are required to be supported. The following table shows that subset and the modes in which a match occurs:

DBGBCR		**Secure**			**Non-Secure**			
[15:13]	**[2:1]**	**OtherP**	**Sys/SVC**	**User**	**Hyp**	**OtherK**	**Sys/SVC**	**User**
000	00		Y	Y			Y	Y
000	01	Y	Y			Y	Y	
000	10			Y				Y
000	11	Y	Y	Y		Y	Y	Y
001	x0	UNPREDICTABLE						
001	01	Y	Y		Y	Y	Y	
001	11	Y	Y	Y	Y	Y	Y	Y
010	00						Y	Y
010	01					Y	Y	
010	10							Y
010	11					Y	Y	Y
011	x0	UNPREDICTABLE						
011	01				Y	Y	Y	
011	11				Y	Y	Y	Y
100	00		Y	Y				
100	01	Y	Y					
100	10			Y				
100	11	Y	Y	Y				
101	xx	UNPREDICTABLE						
110	xx	UNPREDICTABLE						
111	00				Y			
111	x1	UNPREDICTABLE						
111	1x	UNPREDICTABLE						

In this table, OtherP describes the following Secure Privileged modes : IRQ, FIQ, Abort, Undef, Monitor. OtherK describes the following Non-secure Kernel modes: IRQ, FIQ, Abort, Undef

The DBGBVR Meaning field in the DBGBCR is extended by the Virtualization extensions to be bits[23:20]. The encoding of that field is:

BCR[23:20]	Meaning of BVR/BVXR	Function
0000	Unlinked IVA match	IVA == BVR, BXVR ignored
0001	Linked IVA match	IVA == BVR, BXVR ignored
0010	Unlinked Context match (note1)	BVR == ContextID, BXVR ignored
0011	Linked Context match (note1)	BVR == ContextID, BXVR ignored
0100	Unlinked IVA mismatch	IVA != BVR, BXVR ignored
0101	Linked IVA mismatch	IVA != BVR, BXVR ignored
011x	UNPREDICTABLE	
1000	Unlinked VMID match (note2)	BXVR == VMID, BVR must be zero
1001	Linked VMID match (note2)	BXVR == VMID, BVR must be zero
1010	Unlinked Context+VMID match (note1,2)	BXVR:BVR == VMID:ContextID
1011	Linked Context+VMID match (note 1,2)	BXVR:BVR == VMID:ContextID
11xx	UNPREDICTABLE	

Note 1: Context ID matches cannot occur when executing in Non-secure Hyp mode and so these match conditions always fail in Hyp mode.

Note 2: VMID matches can only occur when executing in Non-secure Kernel or Non-secure Non-privileged modes and so these match conditions always fail in secure state or in Hyp mode.

As with the existing architecture, in each of the 001x and 10xx cases:

- If the BRP does support Context ID comparison, breakpoint generation is UNPREDICTABLE
- BCR[8:5] must be 1111
- BCR[19:16] must be 0000
- in the linked cases (0011 and 10x1), BCR[15:13,2:1] must be 00011

13.29 Debug Watchpoint Control Registers (DBGWCR)

DBGWCR[13] is the Hyp match field. This determines whether addresses generated in Hyp mode match for Watchpoints:

0: Match cannot occur in Hyp mode

1: Match can occur in Hyp mode

The DBGWCR privileged mode control field (bits[2:1] has no effect for accesses made in Hyp mode.

Only a subset of the settings of the DBGWCR[15:13, 2:1] bits are required to be supported. The following table shows that subset and the modes in which a match occurs:

DBGWCR		Secure		Non-Secure		
[15:13]	[2:1]	Privileged	User	Hyp	Kernel	User

000	00	UNPREDICTABLE				
000	01	Y			Y	
000	10		Y			Y
000	11	Y	Y		Y	Y
001	x0	UNPREDICTABLE				
001	01	Y		Y	Y	
001	11	Y	Y	Y	Y	Y
010	00	UNPREDICTABLE				
010	01				Y	
010	10					Y
010	11				Y	Y
011	x0	UNPREDICTABLE				
011	01			Y	Y	
011	11			Y	Y	Y
100	00	UNPREDICTABLE				
100	01	Y				
100	10		Y			
100	11	Y	Y			
101	xx	UNPREDICTABLE				
110	xx	UNPREDICTABLE				
111	00			Y		
111	x1	UNPREDICTABLE				
111	1x	UNPREDICTABLE				

13.30 List of Register State to be Added

MRC/MCR p15, 4, Rd, c0, c0, 0	Virtualization Processor ID Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c0, c0, 5	Virtualization Multiprocessor ID Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c1, c0, 0	Hyp System Control Register	Banked-NSHyp only
MRC/MCR p15, 4,Rd, c1, c0, 1	Hyp Auxiliary Control Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c1, c1, 0	Hyp Configuration Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c1, c1, 1	Hyp Debug Control Register	Banked-NSHyp only

MRC/MCR p15, 4, Rd, c1, c1, 2	Hyp Coprocessor Trap Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c1, c1, 3	Hyp System Trap Register	Banked-NSHyp only
MRC/MCR p15, 4,Rd, c1, c1, 7	Hyp Auxiliary Configuration Register	Banked-NSHyp only
MRRC/MCRR p15, 6, Rt1, Rt2, c2	Virtualization Translation Table Base Register	Banked-NSHyp only
MRRC/MCRR p15, 4, Rt1, Rt2, c2	Hyp Translation Table Base Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c2, c1, 2	Virtualization Translation Control Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c2, c0, 2	Hyp Translation Control Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c5, c1,0	Hyp Auxiliary Data Fault Status Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c5, c1,1	Hyp Auxiliary Instruction Fault Status Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c5, c2,0	Hyp Syndrome Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c6, c0, 0	Hyp Data Fault Address Register	Shared with Secure DFAR This encoding Banked-NSHyp only
MRC/MCR p15, 4, Rd, c6, c0, 2	Hyp Instruction Fault Address Register	Shared with Secure IFAR This encoding Banked-NSHyp only
MRC/MCR p15, 4, Rd, c6, c0, 4	Hyp IPA Fault Address Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c10, c2, 0	Hyp Memory Attribute Indirection Register 0	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c10, c2, 1	Hyp Memory Attribute Indirection Register 1	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c12, c0, 0	Hyp Vector Base Address Register	Banked-NSHyp only
MRC/MCR p15, 4, Rd, c13, c0, 2	Hyp Software Thread ID Register	Banked-NSHyp only
Debug register 42	Virtualization ID Sampling Register	Common
Debug registers 144-159	Debug Breakpoint Extended Value Registers	Common

None of the new registers added by the Virtualization Extensions is affected by the CP15SDisable input.

Note1: The registers described as Banked-NSHyp only can be accessed from Hyp mode or from Monitor mode, when SCR.NS==1 in the standard mechanism introduced in the Security Extensions. Accesses to these registers are UNDEFINED from the Kernel Modes within the Non-secure state, or from the Secure Privileged modes when SCR.NS =0

13.31 Access to Banked-NSHyp only registers in Debug State

When in Debug State, access to Banked-NSHyp registers is the same as when not in Debug state, that is the registers can only be accessed when in Hyp mode or in Secure state with SCR.NS==1.

14 NEW INSTRUCTIONS ADDED

14.1 HVC

Encoding T1 Virtualization Extensions

HVC #imm16

15	11	10	4	3	0	15	12	11	0
11110		1111110		imm16<15:12>		1000		Imm16<11:0>	

```
if InITBlock() then UNPREDICTABLE
```

Encoding A1 Virtualization Extensions

HVC #imm16

31	28	27	20	19	8	7	4	3	0
cond		00010100		imm16<15:4>		0111		imm16<3:0>	

```
if cond != 1110 then UNPREDICTABLE
```

14.2 ERET

Encoding T1 Virtualization Extensions

```
ERET   ; Note: this is not a new encoding but is the preferred synonym of SUBS PC, LR, #0 in
           Thumb
```

15	11	10	4	3	0	15	12	11	8	7	0
11110		0111101		(1)(1)(1)(0)		10(0)0		(1)(1)(1)(1)		(0)(0)(0)(0)(0)(0)(0)(0)	

Encoding A1 Virtualization Extensions

ERET

31	28	27	20	19	8	7	4	3	0
cond		00010110		(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)		0110		(1)(1)(1)(0)	

14.3 MSR/MRS Banked Registers

Note: The assembly syntax of these instructions expresses the register and mode within the instruction, while the instruction bit pattern encodes the register and mode in a single 6 bit value, which is split as a 2 bit field (representing the two most significant bits) and a 4 bit field

Encoding T1 Virtualization Extensions

```
MSR <Rm>_<mode>, <Rn>
MSR SPSR_<mode>, <Rn>
MSR ELR_<mode>, <Rn>
```

15	11	10	5	4	3	0	15	12	11	8	7	0
11110		011100		R	Rn		10(0)0		mmmm		(0)(0)mm(0)(0)(0)(0)	

```
SYSm = mm:mmmm
if SYSm[5] == 0 then see MSR Register [1, B6.1.7]
```

Encoding A1 Virtualization Extensions

```
MSR <Rm>_<mode>, <Rn>
MSR SPSR_<mode>, <Rn>
```

```
MSR ELR_<mode>, <Rn>
```

31	28	27	23	22	21	20	19	16	15	12	11	8	7	4	3	0
cond		00010		R	10		mmmm		(1)(1)(1)(1)		(0)(0)mm		0000		Rn	

```
SYSm = mm:mmmm
if SYSm[5] == 0 then see MSR Register [1, B6.1.7]
```

Encoding T1 Virtualization Extensions

```
MRS <Rd>, <Rm>_<mode>
MRS <Rd>, SPSR_<mode>
MRS <Rd>, ELR_<mode>
```

15	11	10	5	4	3	0	15	12	11	8	7	0
11110		011111		R	mmmm		10(0)0		Rd		(0)(0)mm(0)(0)(0)(0)	

```
SYSm = mm:mmmm
if SYSm[5] == 0 then see MRS [1, B6.1.5]
```

Encoding A1 Virtualization Extensions

```
MRS <Rd>, <Rm>_<mode>
MRS <Rd>, SPSR_<mode>
MRS <Rd>, ELR_<mode>
```

31	28	27	23	22	21	20	19	16	15	12	11	8	7	4	3	0
cond		00010		R	00		mmmm		Rd		(0)(0)mm		0000		(0)(0)(0)(0)	

```
SYSm = mm:mmmm
if SYSm[5] == 0 then see MRS [1, B6.1.5]
```

In all cases with SYSm[5] == 1, ELR_<mode>, SPSR_<mode> and <Rm>_<mode> are encoded using the R and SYSm fields:

R=0

	SYSm<4:3>			
SYSm<2:0>	**00**	**01**	**10**	**11**
000	R8_usr	R8_fiq	LR_irq	UNPREDICTABLE
001	R9_usr	R9_fiq	SP_irq	UNPREDICTABLE
010	R10_usr	R10_fiq	LR_svc	UNPREDICTABLE
011	R11_usr	R11_fiq	SP_svc	UNPREDICTABLE
100	R12_usr	R12_fiq	LR_abt	LR_mon
101	SP_usr	SP_fiq	SP_abt	SP_mon

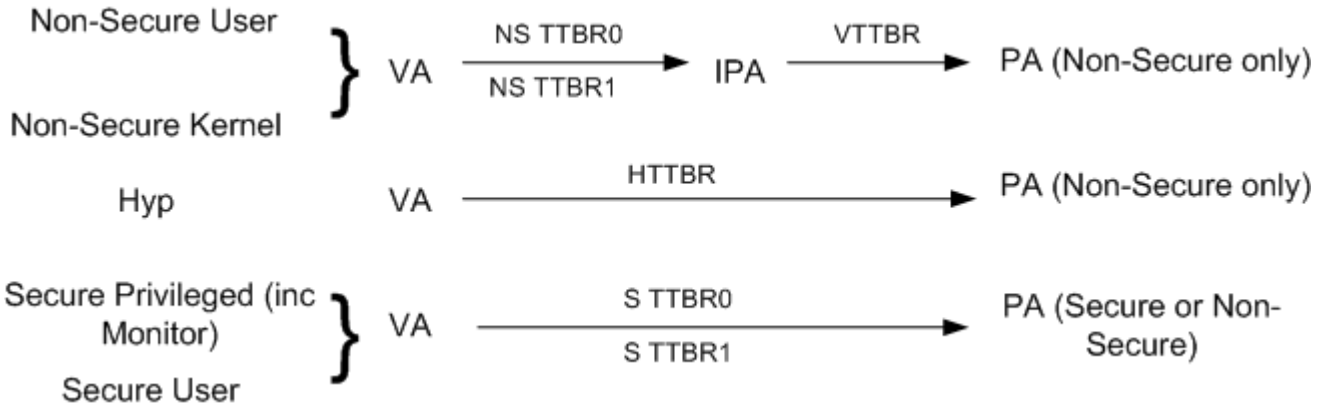
110	LR_usr	LR_fiq	LR_und	ELR_hyp
111	UNPREDICTABLE	UNPREDICTABLE	SP_und	SP_hyp

R=1

	SYSm<4:3>			
SYSm<2:0>	**00**	**01**	**10**	**11**
000	UNPREDICTABLE	UNPREDICTABLE	SPSR_irq	UNPREDICTABLE
001	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE
010	UNPREDICTABLE	UNPREDICTABLE	SPSR_svc	UNPREDICTABLE
011	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE
100	UNPREDICTABLE	UNPREDICTABLE	SPSR_abt	SPSR_mon
101	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE
110	UNPREDICTABLE	SPSR_fiq	SPSR_und	SPSR_hyp
111	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE	UNPREDICTABLE

15 SUMMARY OF THE MEMORY TRANSLATION SYSTEM

The following diagram summarises the overall translation system supported by the ARM architecture with the Virtualization Extensions.



16 REVISED PSEUDO-CODE

16.1 Introduction

The Virtualization Extensions change the pseudo-code for exception entry, by the addition of Hyp mode and the traps associated with Hyp mode. This section defines revised pseudo-code in the light of the Virtualization Extensions.

Five new exceptions are defined from those defined in [1]. These are:

- HVC, caused by the new instruction
- Hyp Trap Exception, caused by executing an instruction that is trapped to Hyp mode as a result of a trap set by a bit in the HCR, HCPTR, HSTR or HDCR. It can only be generated in a Non-secure mode other than Hyp mode, by definition
- Virtual Abort, which can be taken only from Non-Secure Non-Privileged and Kernel modes when HCR.AMO==’1’
- Virtual IRQ, which can be taken only from Non-Secure Non-Privileged and Kernel modes when HCR.IMO==’1’
- Virtual FIQ, which can be taken only from Non-Secure Non-Privileged and Kernel modes when HCR.FMO==’1’

16.2 Exception Entry Helper Functions

16.2.1 HaveVirtExt

The `HaveVirtExt()` function returns TRUE if the Virtualization Extensions are implemented, and FALSE otherwise.

16.2.2 IsAsyncAbort

The `IsAsyncAbort()` function returns TRUE if an abort is asynchronous, and FALSE otherwise.

16.2.3 EnterHypMode

```
// EnterHypMode()
// ==============

EnterHypMode(bits(32) new_spsr_value, bits(32) preferred_exceptn_return, integer vect_offset)
    CPSR.M   = ‘11010’;
    SPSR[]   = new_spsr_value;
    ELR[]    = preferred_exceptn_return;
    CPSR.J   = 0;
    CPSR.T   = HSCTLR.TE;
    CPSR.E   = HSCTLR.EE;
    CPSR.A   = if SCR.EA == ’0’ then ‘1’;
    CPSR.F   = if SCR.FIQ == ’0’ then ‘1’;
    CPSR.I   = if SCR.IRQ == ’0’ then ‘1’;
    CPSR.IT = ‘00000000’;
    BranchTo(HVBAR + vect_offset);
```

16.2.4 EnterMonitorMode

```
// EnterMonitorMode()
// ==================

EnterMonitorMode(bits(32) new_spsr_value, bits(32) new_lr_value, integer vect_offset)
    CPSR.M   = ‘10110’;
    SPSR[]   = new_spsr_value;
    R[14]    = new_lr_value;
    CPSR.J   = 0;
    CPSR.T   = SCTLR.TE;
    CPSR.E   = SCTLR.EE;
    CPSR.A   = '1';
    CPSR.F   = '1';
    CPSR.I   = '1';
    CPSR.IT = ‘00000000’;
    BranchTo(MVBAR + vect_offset);
```

16.3 Exception Entry Functions

16.3.1 TakeDataAbortException

```
// TakeDataAbortException
// ======================

TakeDataAbortException()
    // Determine whether this is an external abort to be trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

    // determine if this should trap to Hyp mode (and distinguish if already there)
    trap_in_hyp = HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' &&
                   CPSR.M == ‘11010’;
    trap_into_hyp = HaveVirtExt() && HaveSecurityExt() && !IsSecure() &&
                  (SecondStageAbort() ||
                  (CPSR.M != ‘11010’ &&
                  (IsExternalAbort() && IsAsyncAbort() && HCR.AMO == ‘1’)||
                  (DebugException() && HDCR.TDE == '1' ))  ||
                  (CPSR.M == ‘10000’ && HCR.TGE == ‘1’ &&
```

```
                (AlignmentFault() || (IsExternalAbort() && !IsAsyncAbort())));
    // if HCR.TGE == ‘1’  and in a Kenrnel mode, the effect is UNPREDICTABLE

    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC plus 4 for Thumb or 0 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 8. For an asynchronous abort, the PC and CPSR are
    // considered to have already moved on to their values for the instruction following
    // the instruction boundary at which the exception occurred.
    new_lr_value = if CPSR.T == '1' then PC+4 else PC;
    new_spsr_value = CPSR;
    vect_offset = 16;
    preferred_exceptn_return = new_lr_value - 8;
    if trap_to_monitor then
        if CPSR.M == '10110' then SCR.NS = '0';
        EnterMonitorMode(new_spsr_value, new_lr_value, vect_offset);
    elsif trap_in_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    elsif trap_into_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);

    else
        // Set SCR.NS to 0 if we are in monitor mode
        if HaveSecurityExt() && CPSR.M == '10110' then SCR.NS = '0';

        CPSR.M = '10111';

        // Write return information to registers, and make further CPSR changes:
        // IRQs disabled, other interrupts disabled if appropriate,
        // IT state reset, instruction set and endianness to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
        CPSR.I = '1';

        if !HaveSecurityExt() || HaveVirtExt() || SCR.NS == '0' || SCR.AW == ‘1’  then
           CPSR.A = '1';
        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
        BranchTo(ExcVectorBase() + vect_offset);
```

16.3.2 TakePrefetchAbortException

```
// TakePrefetchAbortException()
// ============================

TakePrefetchAbortException()
    // Determine whether this is an external abort to be trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

    // determine if this should trap to Hyp mode (and distinguish if already there)
    trap_in_hyp = HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' &&
                   CPSR.M == ‘11010’;
    trap_into_hyp = HaveVirtExt() && HaveSecurityExt() && !IsSecure() &&
                  (SecondStageAbort() ||
                  (DebugException() && HDCR.TDE == '1' &&  CPSR.M != ‘11010’ ) ||
                  (IsExternalAbort() && !IsAsyncAbort() && HCR.TGE == ‘1’
                   && CPSR.M == ‘10000’));
```

```
    // if HCR.TGE == ‘1’  and in a Kenrnel mode, the effect is UNPREDICTABLE

    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 4.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 12;
    preferred_exceptn_return = new_lr_value - 4;

    if trap_to_monitor then
        if CPSR.M == '10110' then SCR.NS = '0';
        EnterMonitorMode(new_spsr_value, new_lr_value, vect_offset);
    elsif trap_in_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    elsif trap_into_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);
    else
        // Set SCR.NS to 0 if we are in monitor mode
        if HaveSecurityExt() && CPSR.M == '10110' then SCR.NS = '0';

        CPSR.M = '10111';

        // Write return information to registers, and make further CPSR changes:
        // IRQs disabled, other interrupts disabled if appropriate,
        // IT state reset, instruction set and endianness to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
        CPSR.I = '1';
        if !HaveSecurityExt() || HaveVirtExt() || SCR.NS == '0' || SCR.AW == ‘1’  then
           CPSR.A = '1';
        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
        BranchTo(ExcVectorBase() + vect_offset);
```

16.3.3 TakeVirtualAbortException

```
// TakeVirtualAbortException
// =========================

TakeVirtualAbortException()

    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC plus 4 for Thumb or 0 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 8. For an asynchronous abort, the PC and CPSR are
    // considered to have already moved on to their values for the instruction following
    // the instruction boundary at which the exception occurred.
    new_lr_value = if CPSR.T == '1' then PC+4 else PC;
    new_spsr_value = CPSR;
    vect_offset = 16;


    CPSR.M = '10111';
    // Write return information to registers, and make further CPSR changes:
    // IRQs disabled, other interrupts disabled if appropriate,
```

```
    // IT state reset, instruction set and endianness to SCTLR-configured values.
    HCR.VA = ‘0’;
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
    BranchTo(ExcVectorBase() + vect_offset);
```

16.3.4 TakeUndefInstrException

```
// TakeUndefInstrException()
// =========================

TakeUndefInstrException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required return
    // address offsets of 2 or 4 respectively.
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 4;

    // if in Hyp mode, then stay in Hyp mode
    trap_in_hyp = HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' &&
                  (CPSR.M == ‘11010’);

    trap_into_hyp = HaveVirtExt() && HaveSecurityExt() && !IsSecure() &&
                  CPSR.M == ‘10000’ && HCR.TGE == ‘1’ ;
    // if HCR.TGE == ‘1’  and in a Kenrnel mode, the effect is UNPREDICTABLE

    return_offset = if CPSR.T == '1' then 2 else 4;
    preferred_exceptn_return = new_lr_value - return_offset;
    if trap_in_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    elsif trap_into_hyp then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);

    else
        // Enter Undefined ('11011') mode, and ensure Secure state if initially in
        // Monitor ('10110') mode. This affects the banked versions of various
        // registers accessed later in the code.
        if CPSR.M == '10110' then SCR.NS = '0';
        CPSR.M = '11011';

        // Write return information to registers, and make further CPSR
        // changes: IRQs disabled, IT state reset, instruction set and
        // endianness to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
        CPSR.I = '1';
        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
```

```
        // Branch to Undefined Instruction vector.
        BranchTo(ExcVectorBase() + vect_offset);
```

16.3.5 TakeSVCException

```
// TakeSVCException()
// ==================

TakeSVCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (the SVC instruction having size 2 or 4 bytes respectively).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 8;

    // if in Hyp mode, then take in Hyp mode
    if HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' && (CPSR.M == '11010') then
        // if in Hyp mode then take in Hyp mode
        preferred_exceptn_return = new_lr_value;
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    elsif HaveVirtExt() && SCR.NS ==  '1' && (CPSR.M ==  '10000') && HCR.TGE ==  '1' then
        // if HCR.TGE is set in User mode, take in Hyp mode as HVC
        preferred_exceptn_return = new_lr_value;
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);
    else
        // Enter Supervisor ('10011') mode, and ensure Secure state if initially
        // in Monitor ('10110') mode. This affects the banked versions of various
        // registers accessed later in the code.
        if CPSR.M == '10110' then SCR.NS = '0';
        CPSR.M = '10011';

        // Write return information to registers, and make further CPSR changes:
        // IRQs disabled, IT state reset, instruction set and endianness
        // to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
        CPSR.I = '1';
        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

        // Branch to SVC vector.
        BranchTo(ExcVectorBase() + vect_offset);
```

16.3.6 TakeHVCException

```
// TakeHVCException()
// ==================

TakeHVCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
```

```
    // the next instruction (with the HVC instruction always being 4 bytes in length).
    ITAdvance();
    preferred_exceptn_return = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    if HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' && (CPSR.M == '11010') then
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 8);
    else
        EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);
```

16.3.7 TakeSMCException

```
// TakeSMCException()
// ==================

TakeSMCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (with the SMC instruction always being 4 bytes in length).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 8;
    if CPSR.M == '10110' then SCR.NS = '0';
    EnterMonitorMode(new_spsr_value, new_lr_value, vect_offset);
```

16.3.8 TakeHypTrapException()

```
// TakeHypTrapException()
// ======================

TakeHypTrapException()
    // HypTrapException is caused by executing an instruction that is trapped to Hyp mode
    // as a result of a trap set by a bit in the HCR, HCPTR, HSTR or HDCR. It can only be
    // generated in a Non-secure mode other than Hyp mode, by definition. Note – where SVC is
    // treated as an HVC as a result of the HCR.TGE, this is NOT trapping of the SVC
    // instruction – see the TakeSVCException pseudo-code

    preferred_exceptn_return = if CPSR.T == '1' then PC-4 else PC-8;
    new_spsr_value = CPSR;
    EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);
```

16.3.9 TakePhysicalIRQException

```
// TakePhysicalIRQException()
// ==========================

TakePhysicalIRQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 24;
```

```
    // Determine whether IRQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.IRQ == '1';
    trap_to_hyp = (HaveVirtExt() && SCR.IRQ == '0' && HCR.IMO == '1' && !IsSecure() ) ||
                  (CPSR.M == '11010');

    if trap_to_monitor then
        if CPSR.M == '10110' then SCR.NS = '0';
        EnterMonitorMode(new_spsr_value, new_lr_value, vect_offset);
    elsif trap_to_hyp then
        preferred_exceptn_return = new_lr_value - 4;
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    else
        // Enter IRQ ('10010')  mode, and ensure Secure state if initially
        // in Monitor mode. This affects the banked versions of various registers
        // accessed later in the code.
        if CPSR.M == '10110' then SCR.NS = '0';
        CPSR.M = '10010';

        // Write return information to registers, and make further CPSR changes:
        // IRQs disabled, IT state reset, instruction set and
        // endianness to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
        CPSR.I = '1';

        if !HaveSecurityExt() || HaveVirtExt() || SCR.NS == '0' || SCR.AW == ‘1’  then
           CPSR.A = '1';

        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

        // Branch to correct IRQ vector.

        if SCTLR.VE == '1' then
            IMPLEMENTATION_DEFINED branch to an IRQ vector;
        else
            BranchTo(ExcVectorBase() + vect_offset);
```

16.3.10 TakeVirtualIRQException

```
// TakeVirtualIRQException()
// =========================

TakeVirtualIRQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 24;


    // Enter IRQ ('10010')  mode.
    CPSR.M = '10010';
```

```
    // Write return information to registers, and make further CPSR changes:
    // IRQs disabled, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';

    CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
    // Branch to correct IRQ vector.

    if SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED branch to an IRQ vector;
    else
        BranchTo(ExcVectorBase() + vect_offset);
```

16.3.11 TakePhysicalFIQException

```
// TakePhysicalFIQException()
// ==========================

TakePhysicalFIQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 28;

    // Determine whether FIQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.FIQ == '1';
    trap_to_hyp = (HaveVirtExt() && SCR.FIQ == '0' && HCR.FMO == '1' && !IsSecure()) ||
                  (CPSR.M == '11010');

    if trap_to_monitor then
        if CPSR.M == '10110' then SCR.NS = '0';
        EnterMonitorMode(new_spsr_value, new_lr_value, vect_offset);
    elsif trap_to_hyp then
        preferred_exceptn_return = new_lr_value - 4;
        EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
    else

        // Enter FIQ ('10001') mode, and ensure Secure state if initially
        // in Monitor mode. This affects the banked versions of various registers
        // accessed later in the code.
        if CPSR.M == '10110' then SCR.NS = '0';
        CPSR.M = '10001';

        // Write return information to registers, and make further CPSR changes:
        // IRQs disabled,
        // other interrupts disabled if appropriate, IT state reset, instruction set and
        // endianness to SCTLR-configured values.
        SPSR[] = new_spsr_value;
        R[14] = new_lr_value;
```

```
        CPSR.I = '1';
        if !HaveSecurityExt() || HaveVirtExt() || SCR.NS == '0' || SCR.AW == ‘1’  then
           CPSR.A = '1';
        if !HaveSecurityExt() || HaveVirtExt() || SCR.NS == '0' || SCR.FW == ‘1’  then
           CPSR.F = '1';

        CPSR.IT = '00000000';
        CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
        CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

        // Branch to correct FIQ vector.

        if SCTLR.VE == '1' then
            IMPLEMENTATION_DEFINED branch to an FIQ vector;
        else
            BranchTo(ExcVectorBase() + vect_offset);
```

16.3.12 TakeVirtualFIQException

```
// TakeVirtualFIQException()
// =========================

TakeVirtualFIQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;
    vect_offset = 28;

    // Enter FIQ ('10001') mode, and ensure Secure state if initially
    // in Monitor mode. This affects the banked versions of various registers
    // accessed later in the code.
    CPSR.M = '10001';

    // Write return information to registers, and make further CPSR changes:
    // IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.F = '1';
    CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to correct FIQ vector.

    if SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED branch to an FIQ vector;
    else
        BranchTo(ExcVectorBase() + vect_offset);
```

16.4 Other significant pseudo-code changes

The following functions are significantly impacted by the changes as part of the Virtualization extensions.

16.4.1 CPSRWriteByInstr()

The third paramenter changes its name to allow the determination that the function has been called as a result of an exception return. This is consistent with its use in the ARM ARM pseudo-code, except for the CPS instruction, which erroneously supplies TRUE as the third parameter – that needs to be changed to be FALSE.

```
// CPSRWriteByInstr()
// ==================
CPSRWriteByInstr(bits(32) value, bits(4) bytemask, boolean is_excpt_return)
    privileged = CurrentModeIsPrivileged();
    nmfi = (SCTLR.NMFI == '1');

    if bytemask<3> == '1' then
        CPSR<31:27> = value<31:27>; // N,Z,C,V,Q flags
        if is_excpt_return then
            CPSR<26:24> = value<26:24>; // IT<1:0>,J execution state bits

    if bytemask<2> == '1' then
        // bits <23:20> are reserved SBZP bits
        CPSR<19:16> = value<19:16>; // GE<3:0> flags

    if bytemask<1> == '1' then
        if is_excpt_return then
            CPSR<15:10> = value<15:10>; // IT<7:2> execution state bits
        CPSR<9> = value<9>; // E bit is user-writable
        if privileged && (IsSecure() || SCR.AW || HaveVirtExt() then
            CPSR<8> = value<8>; // A interrupt mask

    if bytemask<0> == '1' then
        if privileged then
            CPSR<7> = value<7>; // I interrupt mask
        if privileged && (!nmfi || value<6> == '0')  &&
           (IsSecure() || SCR.FW || HaveVirtExt()  then
              CPSR<8> = value<6>; // F interrupt mask
        if is_excpt_return then
            CPSR<5> = value<5>; // T execution state bit
        if privileged then
            if BadMode(value<4:0>) then
                UNPREDICTABLE;
            else
                // Check for attempts to enter modes only permitted in Secure state from
                // Non-secure state. These are Monitor mode ('10110'), and FIQ mode ('10001')
                // if the Security Extensions have reserved it. The definition of
                // UNPREDICTABLE does not permit the resulting behavior to be a security
                //hole.
                if !IsSecure() && value<4:0> == '10110' then UNPREDICTABLE;
                if !IsSecure() && value<4:0> == '10001' && NSACR.RFR == '1' then
                    UNPREDICTABLE;
                // There is no Hyp mode in the secure state, so that is UNPREDICTABLE;
                if SCR.NS == '0' && value<4:0> == '11010' then UNPREDICTABLE;
                // Cannot move into Hyp mode directly from a Kernel mode
                if !IsSecure() && CPSR<4:0> != '11010' && value<4:0> == '11010' then
                    UNPREDICTABLE;
               // Cannot move out of Hyp mode with this function except on an
              // exception return
```

```
            if CPSR<4:0> == ‘11010’ && value<4:0> != ‘11010’ && !is_excpt_return then
                UNPREDICTABLE;

            CPSR<4:0> = value<4:0>; // M<4:0> mode bits
return;
```

17 IDENTIFICATION MECHANISMS

17.1 Processor Feature Register 1 (ID_PFR1)

Bits[15:12] are defined for Virtualization Extensions. Permitted values are:

0b0000 Not Supported

0b0001 Virtualization Extensions architecture supported.

17.2 Memory Model Feature Register 2 (ID_MMFR2)

Bits[19:16] gain an additional value:

0b0100 As 0b0011, and:

Invalidate Unified Hyp TLB entry by MVA

Invalidate entire Non-secure Non-Hyp Unified TLB

Invalidate entire Hyp Unified TLB

17.3 Debug Device ID Register (Debug Device ID Register)

Bits[3:0] gains an additional value:

0b0011 As 0b0010, and:

DBGVIDSR is implemented as register 42

Bits[19:16] defines the Virtualization Extensions to the debug architecture; that is, the extensions described in:

- Section 12.7.1 *Breakpoint and Watchpoint Matching when in Hyp Mode* on page 54
- Section 12.7.2 *BRP extensions for VMID matching* on page 54
- Section 12.8 Distinguishing Stage 1 and Stage 2 Data Aborts in Debug State on page 54
- Section 12.9 Debug Vector Catch Register on page 54

0b0000 The Virtualization Extensions to the Debug architecture are not implemented

0b0001 The Virtualization Extensions to the Debug architecture are implemented